Visual, Interactive Deep Model Debugging: Supporting AI Development and Explainability

Doctoral thesis for obtaining the

academic degree Doctor of

Natural Sciences (Dr. rer. nat.)

submitted by

Thilo Spinner

at the

Universität Konstanz

Faculty of Sciences

Department of Computer and Information Science

Konstanz, 2024

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-2-gejp1cwe65xm0

Dissertation der Universität Konstanz

Tag der mündlichen Prüfung: 24. Mai 2024

1. Referent: Prof. Dr. Daniel A. Keim

2. Referentin: Prof. Dr. Mennatallah El-Assady

Abstract

Despite the significant advancements in deep learning, understanding the inner workings of such models remains a considerable challenge. While this opacity hinders trustworthiness, reliability, and fairness, it also leaves developers unable to identify and mitigate model defects effectively. Methods for explainable artificial intelligence (XAI) have been proposed to mitigate these issues but remain underutilized due to their limited accessibility within typical model development workflows.

To address these issues, we first introduce explAIner, a visual analytics framework that structures the model explanation process and seamlessly integrates XAI techniques into the developer's workflow to improve model understanding, diagnosis, and refinement. The explAIner system enables developers to quickly identify issues with the model's predictions by applying state-of-the-art XAI methods in a comprehensive interface.

While XAI techniques can expose shortcomings in the model's behavior, they often fail to draw connections between errors and their root causes in the model architecture, leaving developers reliant on guesswork and trial and error. Addressing this gap, we introduce iNNspector, a visual interactive framework that establishes theoretical foundations and employs novel mechanisms for the debugging of deep learning models. By correlating architecture and data, the iNNspector system aids developers in pinpointing and rectifying model flaws by providing interactive techniques to explore architectural entities and apply tools to them to investigate underlying data.

The challenges of model debugging shift when dealing with Large Language Models (LLMs), where transformers primarily define the model architecture. Instead, recent advancements suggest that data quality substantially influences LLM performance, moving the debugging focus from the architecture toward the model's inputs and outputs. In existing interfaces, when generating text, the model's outputs are typically presented as running text, concealing the underlying search process, hindering the understanding of uncertainties in the outputs, and omitting viable alternatives. To address this, we present generAItor, which visually unfolds the beam search process, empowering users and computer linguists to comprehend, explore, and refine LLM outputs.

Overall, this thesis contributes innovative visual approaches to assist model developers in understanding and debugging machine-learning models. It presents techniques for enhanced explanation, systematic exploration, and interactive engagement with the model's architecture and decision-making processes.

Zusammenfassung

Trotz erheblicher Fortschritte im Bereich Deep Learning bleibt die Erklärung tiefer neuronaler Netze eine Herausforderung. Neben den damit einhergehenden Problemen, wie zum Beispiel mangelnder Vertrauenswürdigkeit oder Fairness, sind EntwicklerInnen dadurch oft nicht in der Lage, Modellfehler zu identifizieren und zu beheben. Verschiedene Methoden zur Erklärbarkeit künstlicher Intelligenz (XAI) versuchen, diese Probleme zu lösen, sind aber nur unzureichend in den Entwickler-Worfklow integriert.

Mit explAIner stellen wir daher ein Visual Analytics Framework vor, das den Prozess der Modellerklärung strukturiert und XAI-Techniken nahtlos in den Workflow von EnwicklerInnen integriert, um so das Verständnis, die Diagnose und die Verbesserung von Modellen zu erleichtern. Das explAIner-System setzt die theoretischen Grundlagen des Frameworks um und stellt moderne XAI-Methoden in einer übersichtlichen Benutzeroberfläche zur Verfügung.

Während XAI-Techniken Probleme im Modellverhalten zeigen können, scheitern sie häufig daran, Verbindungen zwischen den Fehlern und ihren Ursachen in der Modellarchitektur herzustellen. EntwicklerInnen müssen sich dann auf Vermutungen, ihre Erfahrung oder bloßes Herumprobieren verlassen, um ein Modell zu verbessern. Mit iNNspector stellen wir daher ein visuelles, interaktives Framework vor, das theoretische Grundlagen zur Fehlersuche in Deep-Learning-Modellen etabliert. Das iNNspector-System hilft EntwicklerInnen, Fehler im Modell zu lokalisieren und zu beheben, indem es Werkzeuge zur Exploration von Modellarchitektur und Daten bereitstellt.

In großen Sprachmodellen (LLMs) verschieben sich die Herausforderungen von der Modellarchitektur zu den Ein– und Ausgaben des Modells. So legt aktuelle Forschung nahe, dass die Datenqualität einen erheblichen Einfluss auf die Leistung von LLMs hat. Dadurch, dass in existierenden Systemen zur Textgenerierung lediglich der finale Text gezeigt wird, gehen wichtige Informationen, wie z.B. Unsicherheiten oder alternative Ausgaben, verloren. Wir stellen daher generAItor vor, ein System, das den Suchbaum von Sprachmodellen visuell offenlegt und es BenutzerInnen und ComputerlinguistInnen ermöglicht, die Ausgaben von LLMs zu verstehen, zu explorieren und zu verbessern.

Zusammenfassend präsentiert diese Dissertation innovative visuelle Ansätze, um ModellentwicklerInnen beim Verstehen und Debuggen neuronaler Netze zu unterstützen. Dabei präsentiert sie Techniken, um Modelle zu erklären, zu explorieren und interaktiv mit der Modellarchitektur und den Entscheidungsprozessen zu interagieren.

Acknowledgements

First and foremost, I would like to thank my supervisor, Daniel Keim, for letting me be part of his fantastic working group and giving me the opportunity to pursue my own ideas and research this exciting topic. Thank you, Daniel, for always having an open door and helping me with your experience and advice. I would also like to thank my colleague and advisor Mennatallah El-Assady, who supported me throughout my whole doctoral studies. Thank you, Menna, for your outstanding competence and foresight, for the many valuable discussions, for everything I could learn from you, and also for constantly pushing me beyond my limits.

A big thank-you goes to all the colleagues and co-authors I have had the honor to work with. Here, I would specifically like to mention Hanna Schäfer, Michael Blumenschein, and Rita Sevastjanova, who have particularly supported me and my work. Many thanks go to Fabian Sperrle, Wolfgang Jentner, and the DBVIS Support Team, who provided our chair with an excellent infrastructure and were available all day and night to help. Thanks also to my colleagues from the Visual Computing group, who inspired me to pursue a PhD, and, in particular, to Oliver Deussen, who facilitated the start of my PhD in his group and supported me throughout the entire process.

Special thanks go to my friends, particularly to Lisa, David, Natalie, and Benni. Thank you for celebrating successes with me, supporting me during tough times, and for the many beautiful memories we share.

Thanks to my family, who have always believed in me and had my back. Dear Mom, Dad, and Ivonne, I know that you're always there for me.

Lastly, I would like to thank my wife, Teresa, whose support I could not have written this thesis without. Thank you, Teresa, for taking care of me and my worries, for your patience and your love. Even when the paper deadline is at 2 AM, I know you are there, waiting for me. Thank you for being part of my life.

Danksagungen

Zuallererst möchte ich meinem Betreuer Daniel Keim dafür danken, dass er mich in seine großartige Arbeitsgruppe aufgenommen hat und mir die Möglichkeit gegeben hat meine eigenen Ideen zu verfolgen und an diesem spannenden Thema zu forschen. Danke, Daniel, dass du immer ein offenes Ohr für mich hattest und mir mit deinem Rat und deiner Erfahrung zur Seite gestanden hast. Außerdem möchte ich mich bei meiner Kollegin und Betreuerin Mennatallah El-Assady bedanken, die mich während meiner gesamten Promotionszeit unterstützt hat. Danke, Menna, für deine herausragende Kompetenz und Weitsicht, für die vielen wertvollen Diskussionen, für alles, was ich von dir lernen durfte und auch dafür, dass du mich immer wieder über meine Grenzen hinaus gebracht hast.

Danke auch an alle Kollegen und Co-Autoren, mit denen ich zusammenarbeiten durfte. Besonders hervorheben möchte ich hier Hanna Schäfer, Michael Blumenschein und Rita Sevastjanova, die mich und meine Arbeit in besonderem Maße unterstützt haben. Vielen Dank an Fabian Sperrle, Wolfgang Jentner und das DBVIS-Support-Team, die unserem Lehrstuhl eine herausragende Infrastruktur zur Verfügung gestellt haben und zu jeder Tag– und Nachtzeit verfügbar waren. Danke auch an die Kollegen am Lehrstuhl für Computergrafik, die mich auf die Idee gebracht haben, eine Promotion zu beginnen. Hier gilt ein besonderer Dank Oliver Deussen, der mir ermöglicht hat, meine Promotion in seiner Arbeitsgruppe zu beginnen und mich während der gesamten Zeit fachlich unterstützt hat.

Ein besonderer Dank gilt meinen Freunden, insbesondere Lisa, David, Natalie und Benni. Danke, dass ihr Erfolge mit mir gefeiert und mich in schwierigen Zeiten aufgefangen habt und für die vielen schönen Erinnerungen, die ich mit euch teilen darf.

Danke an meine Familie, die immer an mich geglaubt und mir den Rücken freigehalten hat. Liebe Mama, lieber Papa, liebe Ivonne: Es ist schön zu wissen, dass ihr immer für mich da seid.

Zuletzt möchte ich mich bei meiner Frau Teresa bedanken, ohne deren Unterstützung ich diese Arbeit nicht hätte schreiben können. Danke, Teresa, dass du dich um mich und meine Sorgen gekümmert hast, für deine Geduld und deine Liebe. Auch wenn die Paper-Deadline nachts um zwei ist, weiß ich, dass du auf mich wartest. Danke, dass du Teil meines Lebens bist.

Contents

Abstract **iii**

Acknowledgements **vii**

1 Introduction **1**
- 1.1 Motivation and Research Gaps 1
- 1.2 Research Objectives 2
- 1.3 Thesis Outline and Contributions 4
- 1.4 Publications and Contribution Clarifications 6

2 Related Work **9**
- 2.1 Explainable Artificial Intelligence 9
  - 2.1.1 Algorithm-Centered XAI 9
  - 2.1.2 Human– and Application-Centered XAI 12
  - 2.1.3 Conceptual Work on XAI 14
- 2.2 Debugging of Deep Learning Models 15
  - 2.2.1 Automated Machine Learning 16
  - 2.2.2 Neural Network Visualization 17
  - 2.2.3 Systems and Toolkits 18
- 2.3 Explainable Interactive Language Modeling 19
  - 2.3.1 Language Modeling 19
  - 2.3.2 Language Model Explainability 20
  - 2.3.3 Beam-Search-Tree-Based Visualizations 20
  - 2.3.4 Semantic Similarity 21
  - 2.3.5 Controlled Text Generation 22
  - 2.3.6 Bias Analysis 23

3 explAIner — Explainable and Interactive Machine Learning **25**
- 3.1 Introduction 25
- 3.2 Conceptual Framework for Interactive and Explainable Machine Learning 28
  - 3.2.1 XAI Pipeline 29
  - 3.2.2 Global Monitoring and Steering Mechanisms 32
  - 3.2.3 Use Cases 33
- 3.3 The explAIner System for Interactive and Explainable Machine Learning 35
  - 3.3.1 Understanding 37

3.3.2 Diagnosis 38
3.3.3 Refinement 39
3.3.4 Provenance Tracking and Reporting 40
3.4 Evaluation 41
3.4.1 User Study 41
3.4.2 User Feedback 42
3.5 Discussion 45
3.5.1 Take-Home Messages 45
3.5.2 Limitations and Future Work 46
3.6 Conclusion 48

4 iNNspector — Visual Deep Model Debugging **49**
4.1 Introduction 49
4.2 Capturing the Need For Systematic Model Debugging 53
4.2.1 Research Gap and Open Challenges 53
4.2.2 Capturing Developer Requirements 55
4.2.3 Capturing Developer Workflows and –Challenges 57
4.3 Conceptual Framework for the Systematic Debugging of DL Experiments 60
4.3.1 Capturing the Data Space of Deep Learning Experiments 60
4.3.2 Structuring Design Dimensions for Representation and Interaction 64
4.3.3 Navigating the Data Space through Global Mechanisms 67
4.4 The iNNspector System for Systematic Model Debugging 71
4.4.1 Application Walkthrough 72
4.4.2 Design Rationales 80
4.4.3 Developer Workflow 81
4.4.4 System Architecture & Implementation 82
4.5 Evaluation 85
4.5.1 Use Cases 85
4.5.2 Expert User Study 89
4.6 Discussion 95
4.6.1 Take-Home Messages 95
4.6.2 Limitations and Future Work 96
4.7 Conclusion 98

5 generAItor — Explainable Language Model Predictions **101**
5.1 Introduction 102
5.2 Problem Characterization 104
5.2.1 Users and Stakeholders 104
5.2.2 Challenges 105
5.2.3 The Tree-in-the-Loop Approach 106
5.2.4 User Tasks 107

5.3 Tree Visualization and Model Configuration 109
5.3.1 Beam Search Tree 109
5.3.2 Model Prompting and Configuration 110
5.3.3 Tree Exploration and Explainability 111
5.4 Text Generation, Comparison, and Model Adaptation 112
5.4.1 Text Generation and BST Adaptation 113
5.4.2 Comparative Analysis 116
5.4.3 Model Adaptation 118
5.5 Natural Language Generation Pipeline 119
5.6 Evaluation 120
5.6.1 Case Study 120
5.6.2 Evaluation of Usability and Usefulness 123
5.6.3 Quantitative Evaluation of Model Adaptation 126
5.6.4 Quantitative BST Evaluation 127
5.7 Application: Linguistic Prompting Challenges 130
5.7.1 Identifying Prompting Challenges 130
5.7.2 Applying generAItor to Prompting Challenges 132
5.8 Discussion 138
5.8.1 Rationales of Our BST-Based Approach and Take-Home Messages . 138
5.8.2 Limitations and Future Work 139
5.9 Conclusion 141

6 Conclusion and Future Work **143**
6.1 Summary and Contributions 143
6.2 Broader Impact and Future Research Perspectives 146

List of Figures **149**

List of Tables **150**

List of Listings **150**

List of Listings **150**

List of Abbreviations **151**

Bibliography **153**

1 Introduction

This chapter introduces the topic of this thesis, motivates the research question, and outlines the research objectives and challenges this thesis aims to address. It also provides an overview of the thesis structure, summarizes its contributions, lists the publications that form the thesis, and clarifies the author's contributions to the publications.

1.1 Motivation and Research Gaps

Deep learning (DL) has revolutionized numerous fields with its remarkable ability to generalize and learn complex patterns without explicit feature engineering. However, this strength is also its Achilles' heel; the lack of interpretable features and step-by-step algorithmic reasoning leaves the decision-making process of deep learning models opaque to humans. This *"black box"* nature of deep learning models poses significant challenges for their application in various domains. Often discussed in this context are safety-critical applications, such as autonomous driving [12], or situations where decisions must be justified, such as in the medical domain [13]. Also, recent studies have shown that deep learning models are susceptible to bias and discrimination, which raises concerns about fairness [14]. Besides these challenges to the applicability of deep learning models in real-world scenarios, the lack of transparency and the inability to understand their decision-making process hinder model development and refinement.

Tackling these challenges, the field of explainable artificial intelligence (XAI) emerged, aiming to make the decision-making process of deep learning models more transparent. In the last years, various XAI methods have been proposed to explain the predictions of black-box models [15]. However, despite these advancements, there is an evident lack of accessibility to these methods for non-experts and missing integration into the deep learning workflow.

Besides this missing accessibility, existing XAI methods are often insufficient for deep learning developers for two reasons. First, local methods that provide explanations for single dataset samples are not suitable for reasoning over unseen data. This limitation makes it impossible to predict the model's behavior for new data points, leading to unforeseen behavior in real-world applications, e.g., in autonomous driving [16]. Second, the explanations provided by XAI methods often lack a connection to the underlying

architecture and parameters of the model. This limitation makes it difficult for developers to implement the necessary changes to the model to resolve the issues identified by the explanations. These challenges suggest that a closer look into the model's inner workings is essential to enable developers to effectively diagnose and refine their models.

The emerging field of deep learning debugging aims to address these challenges by providing methods to analyze and understand the inner workings of deep learning models [17]. While some early approaches have been proposed, they still lack a structured process for debugging and do not offer a holistic approach to understanding and rectifying the issues within DL models.

With the recent advancements in language modeling, we are entering a new era of deep learning, bringing with it a new set of challenges for explainability. In the past, the determining factor for the performance of a deep learning model was its architecture. With the advent of the attention mechanism and language transformers [18], the architecture is mostly established, and the focus has shifted to the training data and the sampling process of its outputs [19]. However, the accessibility of the outputs of these models and the explainability of their sampling processes remain largely unexplored.

This thesis tackles these challenges by proposing three visual, interactive approaches, formalizing the process of model explanation, model debugging, and output explanation for large language models (LLMs). The first approach, explAIner, provides a visual analytics (VA) framework for the explainability of black-box machine learning models. It structures the explainability process in a conceptual framework and implements the proposed framework in the explAIner system. The second approach, iNNspector, captures the design space of deep learning experiments and establishes mechanisms guiding the development of debugging systems for deep learning models. It implements these mechanisms in the iNNspector system. The third approach, generAItor, provides a visual interface for the explanation and adaptation of the outputs of large language models. It implements a novel visual, interactive representation of the sampling process of large language models and provides mechanisms to directly interact with the model's outputs. In summary, this thesis aims to advance the field of deep learning development by formalizing the design spaces of model explainability and model debugging and proposing novel visual, interactive systems to address the identified gaps.

1.2 Research Objectives

This thesis aims to make the model development process more informed and transparent by providing approaches for the explainability and debugging of deep learning models. Thereby, the thesis focuses on visual, interactive approaches since they can provide

humans access to and control over explainability methods, the models themselves, and their predictions. Overall, it aims to answer the following research question:

How to enable effective explainability and debugging of deep learning models using interactive, visual interfaces?

To answer this central question, the thesis tackles the following challenges.

(1) General Model Explainability — The landscape of explainable machine learning is vast and scattered, exhibiting a variety of methods and implementations, each with different dependencies and requirements. Also, interactive machine learning (IML) has stakeholders with various backgrounds and expertise, such as model developers, decision-makers, and domain experts. Therefore, formalizing the IML process and establishing a comprehensive framework for model explainability is an essential first step to answering the central research question. While this challenge applies to all machine learning models, the thesis focuses explicitly on deep learning models, which pose additional challenges due to their black-box nature.

(2) Debugging of Traditional Deep Learning Models — To develop and refine deep learning models, the gap between the outcome of explainability methods and the model's inner workings must be bridged. This can be achieved by providing mechanisms to debug deep learning models complementary to the debugging of traditional software systems. Here, the stakeholders are primarily model developers who need to understand the interplay between the model's architecture, the training data, and the model's predictions. Similar to the explainability process, the debugging process is currently under-specified, hindering the development of systems enabling purposeful debugging of deep learning models. Therefore, formalizing the debugging process and establishing mechanisms to guide the development of debugging systems is needed.

(3) Debugging of Large Language Models — The recent advancements in language modeling have led to a new set of challenges regarding their explainability and debugging. With large language models, the focus for model improvement shifts from the architecture to the training data and the fine-tuning process. Also, new groups of stakeholders are suddenly coming in direct contact with deep learning models. First, with the performance of large language models coming closer to human performance, linguistic experts are needed to assess the model's capabilities and limitations. Second, with LLMs now being available to the public, non-experts need to be able to understand the model's general functioning and interact with the model. These shifts require a new approach to explainability, focusing on the outputs of the model and the sampling process. Therefore, new visual interactive approaches are needed to enable linguistic experts to identify and investigate linguistic phenomena in the model's outputs and to provide explainability and accessibility for interested non-experts.

This thesis presents three visual, interactive approaches to overcome these challenges and collectively answer the central research question. The theoretical considerations and frameworks presented in this thesis serve as guidelines for developing visual, interactive systems for the explainability and debugging of deep learning models. The presented system implementations demonstrate how the proposed approaches can be applied to real-world scenarios.

1.3 Thesis Outline and Contributions

Chapter 1 — The introduction identifies research gaps, motivates the thesis topic, and outlines the research question and challenges addressed in this thesis. It also provides an overview of the thesis structure, summarizes the contributions, lists the publications that form the thesis, and clarifies the author's contributions to the publications.

Chapter 2 — This chapter provides an overview of the related work in the fields of explainable machine learning, interactive machine learning, and deep learning debugging. It also discusses recent developments and the state-of-the-art (SOTA) in large language models and approaches to their explainability and debugging.

Chapter 3 — The chapter presents the explAIner framework and –system for the explainability of black-box machine learning models. The primary contributions of this chapter are (1) a conceptual framework that structures the explainability process, provides a taxonomy of explainability methods, and proposes an XAI pipeline for the integration of explainability methods into the deep learning workflow. Furthermore, (2) the explAIner system, which implements the proposed framework and provides a visual interface for the explainability process. Finally, (3) the chapter presents a user study that evaluates the explAIner system by qualitatively assessing the explAIner approach and its influence on the deep learning workflow.

Chapter 4 — The chapter presents the iNNspector framework and –system for the systematic debugging of deep learning models. The main contributions of the chapter are (1) a conceptual framework that categorizes the different types of data generated in deep learning experiments and proposes mechanisms to explore them. The chapter also introduces (2) the iNNspector system, which implements the proposed framework by hooking into the model development workflow and providing a visual interface for the systematic debugging of deep learning models. Finally, (3) the chapter presents a user study that evaluates the iNNspector system by qualitatively and quantitatively assessing the iNNspector approach and its influence on the deep learning workflow.

Chapter 5 — The chapter presents the generAItor system for accessible large language model's outputs, leveraging the beam search tree to explain and adapt the outputs of large

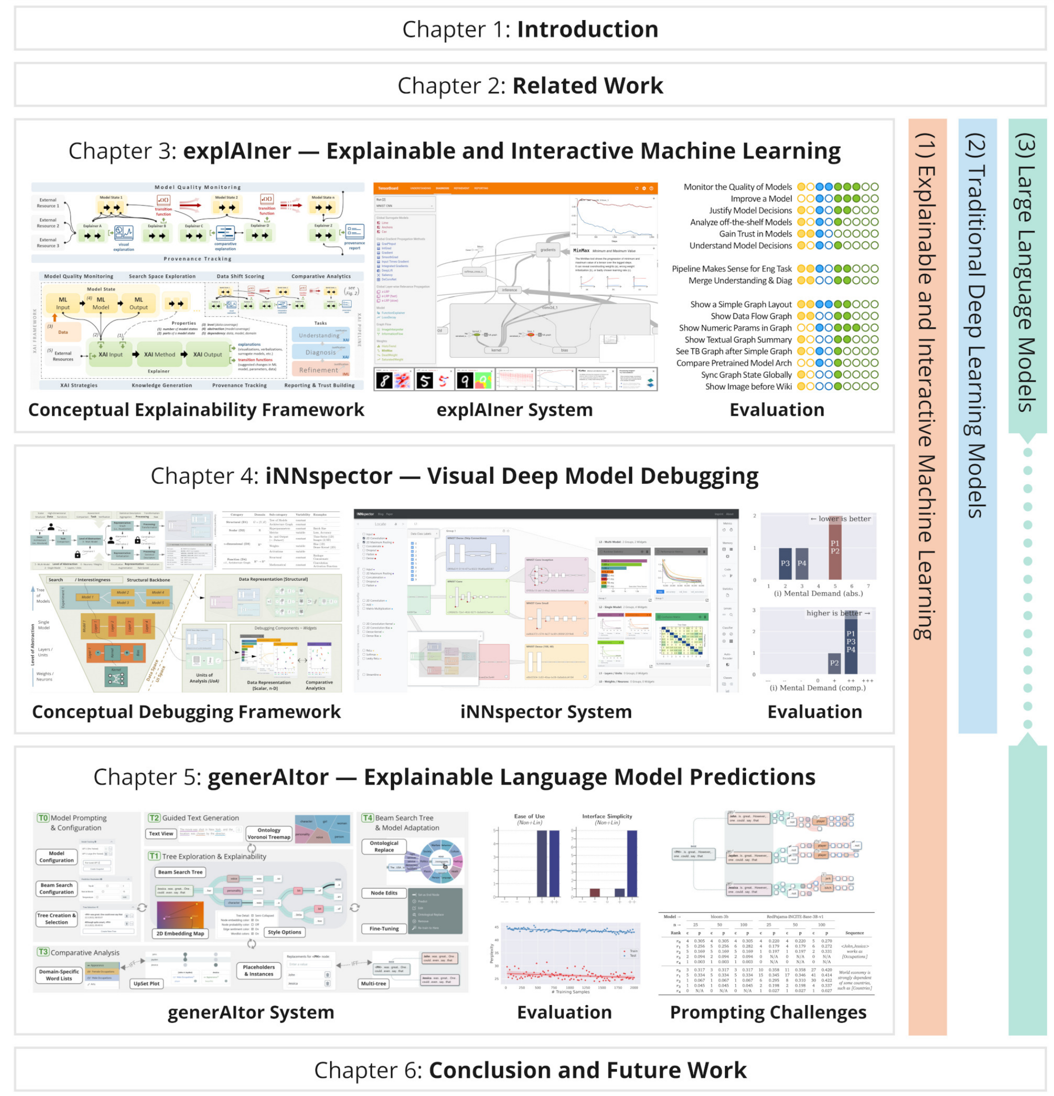


Figure 1.1: Overview of the thesis structure. After the introduction to the topic (chapter 1), the thesis provides an overview of the related work (chapter 2). The following main chapters present the three approaches to the explainability and debugging of deep learning models (chapters 3 to 5). Finally, the thesis concludes with a summary of the main contributions and an outlook on future research directions (chapter 6).

language models. The primary contributions of this chapter are (1) a comprehensive analysis of the challenges in explainability, controllability, and adaptability within the scope of various text generation tasks. It further introduces (2) the novel tree-in-the-loop visual analytics technique, which addresses these challenges through an interactive, tree-centric user engagement approach. Additionally, (3) this chapter presents the generAItor system, implementing the tree-in-the-loop technique in a web-based visual analytics workspace, augmenting the beam search tree with purposeful visualizations and interactive widgets. Also, (4) it includes a multifaceted evaluation of the generAItor

technique, comprising case studies that demonstrate its generative and comparative capabilities, a qualitative user study confirming the usability of the implementation, and a quantitative analysis validating the technique's effectiveness in fine-tuning the model to user preferences. Finally, (5) the chapter shows how the generAItor system can be applied to current challenges in the prompting of large language models from a computer linguistic perspective.

Chapter 6 — The conclusion summarizes the main contributions of this thesis and provides an outlook on future research directions. Thereby, it discusses how the presented approaches answer the research question, how they advance the state-of-the-art, and what the key takeaways are. It also discusses the broader impact of the work, as well as current limitations and future research directions.

1.4 Publications and Contribution Clarifications

This section overviews the journal and conference publications that form this dissertation, outlining the distribution of work among the authors and their respective contributions. The publications are arranged in the order of their appearance in this thesis.

[1] **T. Spinner**, U. Schlegel, H. Schafer, and M. El-Assady. "explAIner: A Visual Analytics Framework for Interactive and Explainable Machine Learning". In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 1064–1074.

Contribution clarification — The paper presents a visual analytics framework for the explainability of black-box machine learning models. It structures the explainability process in a conceptual framework and implements the proposed framework in the explAIner system. The work is based on my idea of providing easy access to methods for model explainability in a comprehensive interface. Mennatallah El-Assady contributed her ideas on how explanation methods could be structured and the explainability process could be formalized, which were refined and translated into the conceptual framework in extensive discussions among the authors. Udo Schlegel summarized the state of the art in explainability methods. Hannah Schäfer took an advisory role, helped conduct the user studies, and provided valuable feedback on the paper. I led the project, coordinated the writing, implemented most of the explAIner system, and performed the user studies. I also wrote the majority of the paper and revised all paper sections several times. Therefore, I use the paper's text without citation marks in this thesis, except for section 3, "Conceptual Framework," which was mainly authored by Mennatallah El-Assady. Since the respective section's content also contains my ideas and is an important foundation for the explAIner system, I rephrase its text in my own words and cite the paper accordingly in this thesis.

[2] **T. Spinner**, D. Fürst, and M. El-Assady. "iNNspector: Visual, Interactive Deep Model Debugging". 2024. arXiv: 2407.17998.

Contribution clarification — The paper structures the process of deep model debugging in a conceptual framework and presents the iNNspector system for systematic model debugging. The paper is based on my own thoughts and ideas. Mennatallah El-Assady took an advisory role, helped shape the ideas through several discussions and design iterations, and provided valuable feedback on the paper and the iNNspector system. Daniel Fürst aided in the implementation of the iNNspector system. Since I wrote the paper entirely on my own, I use the paper's text without citation marks in this thesis.

[3] **T. Spinner**, R. Kehlbeck, R. Sevastjanova, T. Stähle, D. A. Keim, O. Deussen, and M. El-Assady. "generAItor: Tree-in-the-Loop Text Generation for Language Model Explainability and Adaptation". In: *ACM Transactions on Interactive Intelligent Systems* 14.2 (2024).

Contribution clarification — The publication presents the generAItor system for explaining and adapting the outputs of large language models. The paper is based on my initial idea to use a visual representation of the beam search tree to make the sampling process of large language models accessible to the user. The final system design is the result of several design iterations and extensive discussions among the authors. I led the project, coordinated the writing, and implemented the majority of the generAItor system, with Tobias Stähle and Rebecca Kehlbeck also contributing to the implementation. Particularly, Rebecca Kehlbeck developed and implemented the ontology Voronoi treemap, the ontological replace functionality, and the UpSet plot visualization, including the underlying backend to support these features. Consequently, she authored the three respective paragraphs in sections 5.4.1.1 and 5.4.2.1. The rest of the paper was primarily written by me, with contributions from Rita Sevastjanova. Daniel Keim and Oliver Deussen provided feedback on paper drafts. Mennatallah El-Assady took an advisory role and helped shape the system and paper. Since I revised the entire manuscript multiple times, I use the paper's text without citation marks in this thesis, except for the texts authored by Rebecca Kehlbeck, which I rephrase in my own words and cite the paper accordingly.

[4] **T. Spinner**, R. Kehlbeck, R. Sevastjanova, T. Stähle, D. A. Keim, O. Deussen, A. Spitz, and M. El-Assady. "Revealing the Unwritten: Visual Investigation of Beam Search Trees to Address Language Model Prompting Challenges". 2023. arXiv: 2310.11252.

Contribution clarification — The paper identifies state-of-the-art challenges in the prompting of large language models and shows how they can be tackled using the generAItor system. The idea for the paper was developed in discussions between Rita

Sevastjanova, Andreas Spitz, Mennatallah El-Assady, and me. I led the project and wrote the majority of the text, with contributions from Rebecca Kehlbeck and Rita Sevastjanova. Daniel Keim and Oliver Deussen provided feedback on paper drafts, while Andreas Spitz and Mennatallah El-Assady took an advisory role and helped shape the ideas and storyline of the paper. I revised all sections. Therefore, I use the paper's full text without citation marks in this thesis.

Additionally, I contributed to the following publications in chronological order. These publications are not part of this thesis.

[5] **T. Spinner**, J. Körner, J. Görtler, and O. Deussen. "Towards an Interpretable Latent Space: An Intuitive Comparison of Autoencoders with Variational Autoencoders". In: *Proceedings of the IEEE VIS Workshop on Visualization for AI Explainability*. 2018.

[6] M. El-Assady, W. Jentner, R. Kehlbeck, U. Schlegel, R. Sevastjanova, F. Sperrle, **T. Spinner**, and D. Keim. "Towards XAI: Structuring the Processes of Explanations". In: *ACM CHI Workshop on Human–Centered Machine Learning Perspectives*. 2019.

[7] J. Görtler, **T. Spinner**, D. Streeb, D. Weiskopf, and O. Deussen. "Uncertainty-Aware Principal Component Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 822–831.

[8] R. Kehlbeck, R. Sevastjanova, **T. Spinner**, T. Stähle, and M. El-Assady. "Demystifying the Embedding Space of Language Models". In: *Proceedings of the IEEE VIS Workshop on Visualization for AI Explainability*. 2021.

[9] **T. Spinner**, U. Schlegel, M. Schall, F. Sperrle, R. Sevastjanova, B. Gobbo, J. Rauscher, M. El-Assady, and D. A. Keim. "Speculative Execution of Similarity Queries: Real-Time Parameter Optimization through Visual Exploration". In: *Workshop Proceedings of the EDBT/ICDT Joint Conference*. 2021.

[10] M. El-Assady, R. Kehlbeck, Y. Metz, U. Schlegel, R. Sevastjanova, F. Sperrle, and **T. Spinner**. "Semantic Color Mapping: A Pipeline for Assigning Meaningful Colors to Text". In: *IEEE Workshop on Visualization Guidelines in Research, Design, and Education* (2022).

[11] W. Jentner, F. Sperrle, D. Seebacher, M. Kraus, R. Sevastjanova, M. T. Fischer, U. Schlegel, D. Streeb, M. Miller, **T. Spinner**, E. Cakmak, M. Sharinghousen, P. Meschenmoser, J. Görtler, O. Deussen, F. Stoffel, H. Kabitz, D. A. Keim, M. El-Assady, and J. F. Buchmüller. "Visualisierung der COVID-19-Inzidenzen und Behandlungskapazitäten mit CoronaVis". In: *Resilienz und Pandemie: Handlungsempfehlungen anhand von Erfahrungen mit COVID-19*. Ed. by A. Karsten and S. Voßschmidt. Kohlhammer, 2022. Chap. 2, pp. 176–189.

2 Related Work

This chapter presents work related to explainable artificial intelligence and the debugging of deep learning models. The section is structured into three blocks, corresponding to the three focus areas covered in this thesis. Section 2.1 presents work related to explainable artificial intelligence, covering algorithm-centered XAI, human-centered XAI, and application-centered XAI. Section 2.2 presents work related to the debugging of deep learning models, including automated machine learning, neural network visualization, and debugging systems and toolkits. Finally, section 2.3 presents work related to language modeling, visualization and explainability of language models, controlled text generation, and bias analysis.

2.1 Explainable Artificial Intelligence

In this section, we collect important concepts from several survey–, system–, and position papers focusing on explainable artificial intelligence. We structure the XAI approaches into three categories: algorithm-centered, covering the technical perspective, human– and application-centered, covering the human-computer interaction and visual analytics perspectives, and conceptual work, covering the theoretical perspective. We also classify a selection of relevant algorithm-centered XAI techniques according to their properties, shown in table 2.1.

2.1.1 Algorithm-Centered XAI

Summary **Sec. 2.1.1**

Algorithm-centered XAI techniques tackle the black-box nature of neural networks by providing (external) explanations of their predictions. The approaches can be categorized based on attributes like being local or global, intrinsic or post-hoc, and model-agnostic or model-specific. While useful, algorithm-centered XAI techniques are criticized for their limited validity concerning changes in the model architecture.

Explainable artificial intelligence provides techniques to break up the black-box nature of deep learning models, enabling insights into their working [36]. It, therefore, spans

XAI Method	Level		Abstraction		Dependency		
	Global	Local	Low	High	Data	Model	Domain
LIME [20]	○	●	○	●	●	○	○
ANCHORS [21]	○	●	○	●	●	○	○
CAV. [22]	○	●	●	●	●	●	●
e-LRP [23]	○	●	○	●	●	●	●
z-LRP [24]	○	●	○	●	●	●	●
DeepTaylor [25]	○	●	○	●	●	●	●
Saliency [26]	○	●	○	●	●	●	●
Gradient [27]	○	●	○	●	●	●	●
DeepLIFT [28]	○	●	○	●	●	●	●
grad*input [28]	○	●	○	●	●	●	●
Grad-CAM [29]	○	●	○	●	●	●	●
Occlusion [30]	○	●	○	●	●	●	●
SmoothGrad [31]	○	●	○	●	●	●	●
Integrated Grad [32]	○	●	○	●	●	●	●
DeConvNet [33]	○	●	○	●	●	●	●
Node-Link Vis [34]	●	●	●	○	●/○	●	○
Info Flow [35]	●	○	●	○	●/○	●	○
MinMax*	○	○	●	○	○	●	○
HistoTrend*	○	○	●	○	○	●	○
Dead Weight*	●	○	●	○	○	●	○
Saturated Weight*	●	○	●	○	○	●	○

Table 2.1: Properties of XAI methods. **Level** is the data coverage: local (sample) or global (all data). **Abstraction** is the model coverage: high (full model) or low (model parts). **Dependency** specifies necessary inputs for an explainer. (*) are our own implementations.

an essential subtopic of neural network debugging. Adadi and Berrada [37] survey popular XAI techniques and classify them according to mutual properties, such as *local* (explaining single predictions) vs. *global* (explaining the model as a whole). Furthermore, they distinguish between *intrinsic* (the model is inherently explainable, e.g., *decision trees*) vs. *post-hoc* (the model is a black-box and needs external explanation, e.g., *deep neural networks*) and *model agnostic* (the explanation works for different model types) vs. *model specific* (the technique only works for specific model types). This classification is widely accepted in related literature [38]. Guidotti et al. [39] present an extensive state-of-the-art report on XAI, formalizing the problem of explaining black-box models. They classify explanations based on the problem faced, the type of explanator, the black-box it can open, and the type of data used as input by the black-box model. With explAIner, presented in chapter 3, we structure the process of explanation, integrating various XAI techniques into an overarching framework and system.

Guided by Molnar's neatly structured overview on interpretable machine learning [38], in the following, we summarize the state-of-the-art XAI techniques most prominent in deep learning. All presented techniques are post-hoc due to the black-box character of deep learning models. Here, we focus on local, attribution-based techniques. In contrast,

global techniques focus on the explanation of learned features; due to their strong relation to the network architecture, we cover them as a subtopic of section 2.2.2. Local attribution methods can be further divided into model-agnostic and model-specific.

A popular model-agnostic method is Local Interpretable Model-Agnostic Explanations (LIME) [20]. It uses the models' output on a data sample to generate a linear surrogate model explaining feature importance. A similar technique, ANCHORS [21], additionally focuses on the most influential input areas, so-called anchors, to formalize decision rules. Both methods do not consider the underlying model (model-agnostic) but use the sample inputs and outputs of the model (data-dependent) to explain a (local-level) decision boundary generated by the complete model (high abstraction). They can be applied domain independently.

In contrast to LIME and Anchors, Saliency Maps [26] are an example of gradient-based techniques and, therefore, model-specific. They build a visual representation of feature importance by highlighting aspects in each sample as a mask of how the model perceives its input data [39]. There are more techniques to improve the results of saliency maps, such as gradient*input [28], SmoothGrad [31], Integrated Gradients [32], Grad-CAM [29], and DeepLIFT [28]. All of these techniques use a data sample (data-dependent) from the image or text domain (domain-dependent) on artificial neural networks (model-specific) to explain a (local-level) decision generated by the complete model (high abstraction). Additionally, there are two more prominent methods that have similar characterization but slightly different techniques: Layer-wise Relevance Propagation (LRP) [23, 25] abstractly propagates a score from the output to the input to show significant features (e.g., pixel-wise contribution). DeConvNet [33] maps features on pixels to show the reverse activation of convolutional layers.

In contrast to these high-abstraction methods, Concept Activation Vectors (CAVs) [22] operate on concrete network layers. This XAI method enables users to verify *how* their (data-dependent) understanding of a concept's importance (e.g., stripes) is represented in the ANNs (model-specific) prediction (e.g., zebra) for a sample (local-level) image (domain-dependent) in each or all layers (low+high abstraction).

For a classification of prominent *XAI methods* according to their properties, see table 2.1. All reviewed XAI methods are supported by the explAIner framework, and most of them are part of the explAIner system, presented in sections 3.2 and 3.3.

As opposed to the feature-importance-based methods discussed above, counterfactuals [40] probe the decision boundary of a model by finding the smallest possible change in the input to generate a desirable outcome, e.g., to flip the predicted class.

Notably, the described techniques are widely criticized for being fragile [41], prone to manipulations [42], misleading [43], and barely actionable towards architectural changes. iNNspector, presented in chapter 4, tackles these issues by targeting a holistic

view of neural networks, inherently connecting the model's behavior to its architecture. The discussed algorithm-centered XAI techniques can be considered potential tools in iNNspector's overarching debugging framework, spanning a sub-task of neural network debugging.

2.1.2 Human– and Application-Centered XAI

Summary
Sec. 2.1.2

Human-centered XAI focuses on integrating human understanding into the explanation process. Our work aligns with this by making model explanation and –debugging an interactive process. Application-centered XAI facilitates this human involvement through visual analytics systems tailored to explain machine learning models. The strong specialization of existing systems to data, tasks, or model architecture often limits reusability and transferability. This thesis addresses this issue by proposing generalizable mechanisms for model explainability and debugging.

Human-centered explainable AI (HCXAI) [44] strives to fix the shortcomings of algorithm-centered techniques to disregard the human's role in the explanation process. Recently, there has been a surge in calls for explainability that does not only open the black box but also considers the individual and social factors of the human interacting with the black box [45]. The field of HCXAI is currently evolving; recent positions [46], design guidelines [47], frameworks [48], and studies [49] outline possible future directions. Our work takes up the concept of tightly integrating the human into the machine-learning loop; explainability and model debugging inherently are about matching human intentions against the behavior of the algorithm [50].

Closely related to HCXAI are visual analytics techniques for interactive machine learning and model explanation, aiming to boost the model development process through tailored visual interfaces [51]. Visual analytics for IML tightly integrates the user to promote further sensemaking during the model development workflow [52]. In recent years, various visual analytics systems for machine learning have been presented, spanning a wide range from particular application areas to tools covering one or multiple stages of the machine learning workflow [53]. In contrast to the algorithm-centered techniques discussed in section 2.1.1, these systems not only provide visualizations of models and data but integrate the human into the IML loop [51]. Hohman et al. [54] review and structure recent approaches based on the 5 *W*'s and *How*. They identify different entities that can be explained (i.e., *"What?"*), such as computational graph & network architecture [35], learned model parameters [55], individual computational units [56], or neurons in high-dimensional space [57].

During the following review of existing IML/VA systems, we classify the solutions according to how they cover the three tasks of the IML pipeline *understanding*, *diagnosis*, and *refinement* [58], as well as *reporting*.

Understanding — The understanding phase can be interpreted in different ways depending on the target user group. For a model novice, some systems use VA as an educational tool to explain machine learning (ML) concepts. For instance, Harley [34] visualizes changes in an image along with the affected layers of an artificial neural network (ANN). Smilkov et al. [59] also provide an interactive, visual representation of an ANN. Further work offers various ways to explore the graphical representation of deep nerual networks (DNNs), with Wongsuphasawat et al. [35] focusing on the architectural component and Kahng et al. [60] designing a dataflow pipeline of generative adversarial networks (GANs). In contrast to the educational goals of model novices, model users and model developers need to understand the model's inner workings. Rauber et al. [61] focus on this aspect by visualizing the ANN training, as well as both neuron-neuron and neuron-data relationships. Bilal et al. [62] visualize the hierarchical abstraction of CNNs, highlighting the importance of multiple abstraction layers. Representing features with low abstraction, Strobelt et al. [57] explore the inner workings of the hidden cell states, the activation, as well as [63] the attention component of model structures. Based on the lessons learned from these works, we conclude that there is a need for providing tailored model explanations on different model abstraction levels.

Diagnosis — Many VA systems address this gap by focussing on a model's diagnosis in an IML workflow to enable the detection of problems on different abstraction layers. Some systems support a model-agnostic diagnosis by focusing on feature importance [64] or the reaction of the model to real [65] or adversarial input examples [66]. Others focus on specific elements, such as the neuron activation [56], hidden states of a cell [67], or action patterns of reinforcement learning algorithms [68] to allow model-specific diagnosis. Finally, some systems visualize the dataflow [55] and decision paths [69] taken by the model to enable a model diagnosis during the training process.

Refinement — While all these approaches allow for an integrated diagnosis, they fall short of addressing the identified issues in a subsequent refinement step. Therefore, some VA systems go beyond diagnosis and target the refinement of ML models using interactive visualizations. Several works focus on the diagnosis and refinement of *single* ML models [70, 71, 72], while others target *multi-model* visual comparison for refinement [73, 74]. In addition to this distinction, various interactive refinement approaches are used in iterative cycles. For example, Cai et al. [75] present an approach for medical images or El-Assady et al. [76, 77] for topic modeling.

Reporting — Besides, the reporting of results and the tracking of changes are essential elements of IML [78]. Krause et al. [79] and Ming et al. [80] both provide a visual representation of a feature's importance to the model output, while Sevastjanova et

al. [81] support tracking the entire workflow, as a mixed-initiative, active learning system. These approaches show that an XAI framework should go beyond these three IML tasks and incorporate global monitoring and steering mechanisms.

All the reviewed approaches are highly specialized in their use cases and cover only selected phases of the IML workflow. In contrast, with the explAIner framework presented in section 3.2, we propose a pipeline that can cover different pathways through all of the addressed tasks. To aid this pipeline, global monitoring and steering mechanisms can support and guide the overall process of IML.

Furthermore, the specialization of these systems to data, tasks, or models prevents their re-usability and hinders their application in practice. iNNspector, presented in chapter 4, tackles these limitations by proposing mechanisms to make the entirety of data explorable and presenting an extensible system that implements those mechanisms. Bäuerle et al. [82] identify a similar research gap and propose *Symphony*, a framework for composing interactive interfaces for machine learning. While iNNspector is designed as a standalone tool, Symphony integrates into external platforms like Jupyter notebooks. Also, the systems differ in their way of accessing, navigating, and visualizing data, e.g., Symphony provides no mechanisms to access or navigate structural data (c.f., section 4.3.3). In their DL debugging framework *Cockpit*, Schneider et al. [83] gather higher-order information on the gradients during training and visualize them to identify common bugs in data processing, parameter selection, and model architecture. To this end, they define different quantities tracked and computed during training and visualize them in plots called "instruments." While both Cockpit and iNNspector provide plots of internal model dynamics, they differ in their focus. Cockpit provides specialized instruments visualizing advanced internal dynamics during training. In contrast, iNNspector provides a broader range of instruments, covering the entire deep learning workflow, complementing the plots of model metrics with visualizations of model architectures, in- and output data (e.g., images and classification results), and model-internal representations (e.g., kernels or activations).

2.1.3 Conceptual Work on XAI

Recent surveys approaching explainable machine learning from a theoretical perspective identify gaps in explainable AI, such as the need for formalism, interpretability in real-world applications, and explanations for model novices. Furthermore, surveys of visual analytics for machine learning identify the need for a general-purpose framework for interactive machine learning. We tackle this gap with the proposed explAIner framework and system (sections 3.2 and 3.3).

Several surveys and position papers provide a conceptual view of explainable artificial intelligence, striving to structure the field and identify open research questions.

The recent survey by Adadi and Berrada [37] provides an entry point to XAI, covering basic concepts, existing methods, and future research opportunities. However, they identify a lack of formalism in the field and demand "clear, unambiguous definitions," revealing a research gap for a conceptual framework structuring the XAI process, which we intend to fill. While Doshi-Velez and Kim [84] provide definitions for interpretability, they observe a need for real-world applications. Our explAIner framework and subsequent system implementation consider the directions they give for establishing a general and multifaceted model. By summarizing XAI motivations and characteristics, Guidotti et al. [39] present an extensive overview of XAI, especially current methods for the explanation of models, which our framework incorporates to tackle the latest challenges black-box models impose.

Regarding interactive machine learning, recent developments are captured by Jiang and Liu [85], who identify open research questions, including explanations for model novices, as well as global monitoring of the analytical process of explainability. These build the foundation for the monitoring and steering mechanisms of our framework. In the context of visual analytics, Liu et al. [58] structure the IML workflow into the three tasks of understanding, diagnosis, and refinement, which we utilize to structure the process of explainability in our XAI pipeline. In our framework, we focus on deriving synergies from this combination, e.g., by closing the ML loop between diagnosis and refinement. Such synergistic effects have been recently surveyed by Endert et al., who summarize recent advances in the integration of ML into VA [52], such as combining interactive visualization approaches and controllable ML algorithms. Focusing on VA in the field of deep learning, Hohman et al. [54] use an interrogative survey method to categorize recent work according to the six W-Questions [86]. Based on their discussion of research opportunities, we decided to target the three user groups and the four goals (interpretability, debugging, comparing, and education) they identify.

2.2 Debugging of Deep Learning Models

Summary
Sec. 2.2

We identify several subtopics touching the field of deep model debugging, providing a broad spectrum of tools and methodologies, each with its strengths and limitations. In contrast to the approaches presented in this thesis, the state-of-the-art techniques only tackle sub-parts of deep learning experiments, constraining them to specific domains, models, tasks, or stages of the model development workflow.

The work related to the debugging of neural networks can be divided into several subtopics, including techniques for explainable artificial intelligence, automated machine learning, visual analytics for machine learning, and systems for tracking and assessing machine learning experiments. Work on explainable artificial intelligence is extensively covered in section 2.1. Therefore, in the following, we focus on automated machine learning, neural network visualization, and debugging systems and toolkits.

2.2.1 Automated Machine Learning

Summary
Sec. 2.2.1

AutoML techniques aim to automate model building and tuning. The subfields include hyperparameter optimization, meta-learning, and neural architecture search. While these techniques offer automation of model refinement, they still require explanation and debugging for human control and trust-building.

In contrast to the manual model building, –diagnosis, and –refinement process, techniques for automated machine learning (*AutoML*) strive to automate the time-consuming hyperparameter search by systematically probing the search space, as summarized by [87]. They divide the field into three subtopics: *hyperparameter optimization* is the most basic task in AutoML, addressing the search for parameter configurations (e.g., layer capacities, optimization algorithms, regularization) that maximize the model's performance. Here, recently, Bayesian optimization-based approaches [88], multi-fidelity methods [89], and a combination of both [90] proved successful. In contrast, *meta learning*, or *learning to learn*, strives to speed up learning a new task by learning from existing models. In deep learning, prominent examples are transfer learning [91], where models trained on a source task are used as a starting point to learn a target task, and few-shot learning [92], where prior experiences with similar tasks are exploited to train a model on a new task for which only few training samples are available. Finally, *neural architecture search* focuses on the automated search for model architectures [93], making it the most complex task of AutoML. Several search strategies exist; evolutionary algorithms [94] have a long history in the field [95] and still prove successful [96]. More recently, Zoph and Le's advancements using reinforcement learning [97] brought new momentum to the field. Other approaches rely on Bayesian optimization [98] or random search [99].

While AutoML techniques can automate large parts of the model refinement and parameter tuning process, model explanation and systematic debugging is nevertheless relevant. Tracking search space coverage, trust-building, and model verification are essential when the model-building process is left to an opaque optimization algorithm. For instance, to rule out overfitting of the AutoML algorithm [100] or when model explainability is mandatory [36].

2.2.2 Neural Network Visualization

Summary
Sec. 2.2.2

Various tools exist for visualizing neural network architectures and learned representations. Our approaches extend architecture visualizations through information on architectural building blocks (section 3.3.1) and through incorporating a higher-order graph to show relationships between different model generations (*Multi-Model View* in section 4.4.1). Also, we simplify access to learned representations through single-click tools (*Toolbox, Widgets* in section 4.4.1).

With the abundance of network architectures and –types, various approaches to visualize these architectures arose. Patel [101] presents a comprehensive overview of tools to visualize neural networks. With NN-SVG, LeNail [102] provide a tool to generate SVG visualizations in the fully-connected–, LeNet– [103], and AlexNet-styles [104]. TensorSpace.js[1] combines interactive 3D visualizations of the networks with explanations of the model's prediction process. TensorBoard [35, 105] visualizes the computational graph of TensorFlow models down to single operations. In contrast, Netron [106] supports various DL frameworks and generates more abstract, layer-based graph representations. Wang et al. [107] present an interactive genealogy of prevalent network types and –architectures.

Besides or in conjunction with architecture visualization, many approaches focus on the visualization of learned representations, particularly in computer vision [108]. Common are pixel displays of activations [109] or kernels [34, 110], and heat maps [111]. Approaches for feature visualization [112] strive to explain the functionality of individual neurons, channels, or layers through activation maximization, putting it at the junction of feature-based network visualization and global, algorithm-centered XAI. Hohman et al. [113] combine feature visualizations with displays of the network architecture in the form of attribution graphs, associating individual dataset classes with strongly activated subnetworks. While these approaches provide valuable insights into the model's internals, they are often hard to access and require expert knowledge to implement.

The iNNspector framework and –system presented in sections 4.3 and 4.4 combine architecture visualizations of layers and neurons with visualizations of learned representations. For the first time, we extend the visualization of model architectures with a higher-order graph, expressing the relationship between models and differences between model generations. Furthermore, our toolbox-based approach allows single-click visualizations of learned representations, making them easily accessible.

[1] https://tensorspace.org/

2.2.3 Systems and Toolkits

Summary
Sec. 2.2.3

Various systems and toolkits exist for explaining and debugging machine learning experiments. While these systems provide different functionalities for model management, visualization, explanation, and assessment, a holistic view of the deep learning workflow must be included. Our presented frameworks structure the explanation and debugging processes and propose components and mechanisms to guide the development of such systems.

Focussing on the integration of explainability techniques, iNNvestigate [27] provides out-of-the-box implementations of prominent XAI techniques, such as LRP [23] and Integrated Gradients [32]. A similar system is DeepExplain [24], which provides improved algorithms and implementations for LRP and DeepLIFT. While these systems provide a broad collection of different XAI methods, they are limited to specific XAI techniques and do not cover the entire interactive machine learning workflow, which we address with the explAIner framework and –system presented in chapter 3.

Keeping track of experiments is an essential task in the daily machine-learning workflow. Systems like TensorBoard [105] or Weights & Biases [114] provide various tools for model management, –assessment, and –comparison. Furthermore, TensorBoard offers tools for visualizing the architecture and data flow of a model [35], as well as explainability techniques [115]. TensorWatch [116] is a debugging and visualization tool that integrates into Jupyter[2] Notebooks. The Error Analysis toolkit [117] provides tools to identify and diagnose data cohorts with high error rates. Specialized in the field of natural language processing, the Language Interpretability Tool [118] supports model understanding based on what-if-probing.

The discussed techniques and systems cover model debugging concerning specific domains, models, or tasks. In contrast, the iNNspector framework and –system introduced in chapter 4 targets the systematic debugging of deep learning experiments (1) across all stages and (2) over any data arising in the deep learning workflow. To this end, the conceptual framework we present structures the data and design space and provides tools and techniques to cover the spaces accordingly, without introducing constraints to domain, model, or task.

[2]https://jupyter.org/

2.3 Explainable Interactive Language Modeling

In the following, we present work related to explaining large language models. It is structured according to the subtasks of language modeling, semantic similarity, controlled text generation, bias analysis, LM explainability, and beam-search-tree visualization.

2.3.1 Language Modeling

Summary
Sec. 2.3.1

Language models are central in NLP systems. With the transformer architecture, a new generation of language models emerged, which surpasses previous models in performance and efficiency. Transformers are categorized into masked language models (e.g., BERT) for classification and generative models (e.g., GPT) for text generation. Our presented generAItor system uses both types of models, with a focus on the generative capabilities of transformers.

Language models (LMs), as fundamental components of natural language processing (NLP) systems, are probability distributions over word sequences [119]. With the emergence of the transformer architecture [18], there was a paradigm shift away from recurrent neural networks (RNNs) [120] since transformers allow parallel computations which speed up training times and prove superior in capturing long-term dependencies [18]. They use the attention mechanism [121], which directs the focus on important tokens in the input sequence. Nowadays, numerous pre-trained transformer architectures are available for public use [122]. There are different types of transformers, whereby the two main categories are masked– and generative language models.

Masked LMs — BERT [123] is a transformer-based LM that was trained on masked language modeling (i.e., *cloze*) and next-sentence prediction tasks and is commonly fine-tuned for diverse text classification tasks [124]. Due to its pre-training objective, BERT (as well as other masked language models) is not suitable for text generation tasks. We use BERT for masked word prediction in the *ontological replace* functionality of generAItor, described in section 5.4.1.

Generative LMs — Text can be generated using generative transformer models, of which prominent examples are OpenAI's GPT family, including GPT-2 [125], GPT-3 [126], and GPT-4 [127]. They are autoregressive models that were pre-trained on the causal language modeling task, learning to predict the next token in a sequence. See the survey on pre-trained language models for text generation by Li et al. [128] for a broader overview. Particularly, in our work, we use GPT-2, Bloom [129], and RedPajama [130] for text generation; however, the approach is designed to support other transformer-based LMs as well.

2.3.2 Language Model Explainability

Summary
Sec. 2.3.2

Explainability of language models is a recent research field, mostly focusing on the explainability of the inner workings of the model. In contrast, our generAItor work focuses on the explainability of the model's outputs, providing explanations for the model's predictions.

With the rise of large language models, the explainability of their inner workings and the interpretability of their outputs expanded the field of explainable AI. Matching the four categories as proposed by Danilevsky et al. [131], approaches usually use explainability techniques in conjunction with a set of operations to enable explainability and visualization techniques to convey the operations to the user. Examples are visualizing saliency to explain feature importance for local post-hoc [132] or training a surrogate model to allow for taxonomy induction, providing global explanations [133]. As identified by Yuan et al. [53], explanations are needed before, during, and after model building, and it is crucial to identify ways to intuitively convey model outputs to the user and allow for an exploration of model outputs. In the context of visual analytics approaches for the explainability of deep neural networks, Rosa et al. [134] survey common visualization techniques used in visual analytics systems for explainability and identify a lack of tree-based visualization techniques. Our proposed tree-in-the-loop method outlined in section 5.2.3 is based on a representation of the beam search tree and complements it with a set of interactions for example-driven, instance-based investigation of NLP challenges, offering both self-explaining and post-hoc local explanations.

2.3.3 Beam-Search-Tree-Based Visualizations

Summary
Sec. 2.3.3

Beam search is an essential part of the decoding process in language models. Different approaches exist for visualizing the beam search tree, most tailored to tasks such as machine translation or table-to-text generation. Instead, our generAItor approach focuses on the explainability of the model's outputs for text generation and adaptation.

Beam search is an essential part of the decoding process in LMs. Visualizing and using the created beam search tree is, therefore, a possibility to investigate predictions and allow user interaction with the tree. Lee et al. [135] use a basic beam search tree visualization for the task of neural machine translation. Their tool visualizes the beam search decoder with probabilities and allows basic tree manipulation. Also, for machine translation, Seq2Seq-Vis was proposed by Strobelt et al. [63], which focuses on helping the user debug and find errors in the translation result. The user can investigate all steps of the translation pipeline to help improve the translation result for single instances.

For larger document collections, Munz et al. [136] propose a visual analytics system to help identify and correct single instances and propagate corrections for larger document collections. They also visualize the beam search tree and allow basic interactions on the node level to correct translations. Strobelt et al. [137] introduce GenNI, a system for collaborative table-to-text generation by applying user-defined constraints to the beam search tree, guiding the produced outputs.

2.3.4 Semantic Similarity

Semantic similarity assesses the likeness in meaning between text segments. We distinguish two types of approaches: knowledge-based approaches, which use taxonomies and ontologies, and embedding-based approaches, which use embeddings learned from data distributions.

Semantic similarity, a key concept in understanding language models, involves assessing the likeness in meaning between text segments. This section explores the various approaches and tools used to analyze and utilize semantic similarity in language models. We distinguish between knowledge-based (taxonomies and ontologies) and embedding-based approaches (embeddings learned from data distributions).

Word Taxonomies and Ontologies — Leveraging semantic graphs and knowledge bases, such as YAGO and DBpedia, it is possible to infer concept or topic hierarchies via language models [138, 139, 140] or expand existing taxonomies [141, 142]. Methods such as OntoEA [143] align entities by jointly embedding ontologies and knowledge bases. Taxonomies can be used to improve recommender systems [144] and help with entity recognition [145] or translation [145]. WordNet information can be integrated into pre-trained language models for improved sense disambiguation, e.g., ARES [146], or used to build human-readable concept vectors [147]. For generAItor's ontology-based functionalities, discussed in section 5.4.1, we use ARES and BERT embeddings in conjunction to create domain-specific predictions with an ontology graph created from the BabelNet [148] semantic graph.

Embedding Similarity — In language models, each token of the input text is mapped to a high-dimensional vector. Related work has shown that these context-dependent embeddings encode different context/language properties. Although BERT is the most widely analyzed language model so far [149], other transformer models, such as GPT-2, and their produced embedding spaces have also attracted computational linguistics' and visual analytics researchers' attention [150, 151]. Prior research has shown that semantic information, such as word senses and semantic roles, is captured best in the higher layers of transformer models [152, 153, 151]. Thus, these contextualized embeddings are commonly used as features for semantic similarity tasks. As described in section 5.3.1,

in generAItor, we apply a dimensionality reduction technique on embeddings extracted from the used LMs to map the tokens to unique colors based on their coordinates in the two-dimensional space. With this approach, tokens with a semantic similarity get assigned to similar colors [10].

2.3.5 Controlled Text Generation

Summary
Sec. 2.3.5

Controlled text generation guides language models to produce specific outputs using algorithmic or interactive methods. Algorithmic approaches automatically control aspects like sentiment and topic. Interactive methods involve tools for text editing and collaborative generation with human involvement.

Controlled text generation is about directing language models to produce output with specific characteristics. This section reviews the algorithmic and interactive approaches that guide the style and content of generated text.

Algorithmic Approaches — In general, controlling the style and information of natural language generation (NLG) is one of the applications identified by Gatt and Krahmer [154]. One challenge of integrating knowledge into text generation is the automatic steering of the generation in a particular direction. Using plug-and-play language models is one possibility to steer text generation [155]. Concerning pre-trained language models, it is possible to control, e.g., the sentiment [156, 157], keywords [158], or the topic [156]. Frameworks such as FAIR [159] allow the generation of content-controlled text by combining BERT with BART [160]. A larger overview is given in the survey by Zhang et al. [161]. Building on this, many approaches now integrate external resources such as knowledge bases. More details can be found in the survey by Yu et al. [162]. However, these techniques do not allow immediate intervention in the decision process, which we specifically target with our generAItor approach.

Visual Interactive Approaches — Focusing on interactive editing, Du et al. [163] provide interactive suggestions in their tool to achieve high-quality text edits with minimal human effort. Padmakumar and He [164] use a human-in-the-loop approach to replace text segments for the task of creative image captioning. Gehrmann et al. [165] propose an interactive framework that allows users to control generative segments through a process called collaborative semantic inference. Following this, Strobelt et al. [137] create GenNi, an interface for collaborative text generation. They guide the model output using explicitly defined constraints. The user has to know beforehand how he wants to control the model output, as it is impossible to adapt the state during inference. With Wordcraft, Yuan et al. [166] present an interactive interface that allows writers to create stories with the assistance of large language models. Their system lets authors re-write, replace, and auto-generate text, as well as define custom requests to

the language model. In contrast, our generAItor approach enables direct interaction with the model's outputs by exposing predictions and probabilities in the beam search tree.

2.3.6 Bias Analysis

Summary
Sec. 2.3.6

Bias mitigation is one of the most urgent challenges in generative models. Often, methods for bias analysis are based on templates or pre-defined word lists. In contrast, our generAItor approach allows the detection of biases in variable-length sequences and the identification of subtle nuances in the model outputs.

Bias analysis in language models is crucial for ensuring fairness and inclusivity in AI systems. This section examines the current research on detecting and mitigating biases in language models, highlighting the challenges and methodologies in this area.

Current research explores not only what the models learn but also when they fail and which limitations they have, such as different types of biases [167]. For instance, Blodgett et al. [168] present a taxonomy for fairness definitions that machine learning researchers have defined to avoid existing bias in AI systems. Mehrabi et al. [169] define the bias problem specifically in language modeling tasks in a formal way and explore how it has been treated in related work regarding their detection and correction.

In masked language models, the detection of bias is typically done by applying templates or pre-defined word lists. For instance, the Word Embedding Association Test (WEAT) [170] measures the association between two target word sets (e.g., male pronouns and, e.g., female pronouns) based on their cosine similarity to words from two attribute sets (e.g., terms related to science or art) to make conclusions about encoded biases. Liang et al. [171] show that the analysis of biases in text generation can be more nuanced, e.g., biases can arise during the generation of any token [172]. Alnegheimish et al. [173] find that bias "evaluations are very sensitive to the design choices of template prompts." According to the authors, the use of template-based prompts tends to evoke biases from the model's default behavior rather than reflecting the actual correlation between gender and profession, analyzed in their work. Thus, we propose a tree-based approach for comparative, exploratory bias analysis, allowing the detection of biases in variable-length sequences and the identification of subtle nuances in the model's predictions, which is presented in section 5.4.2. For a detailed case study, showcasing the benefits of generAItor's comparative functionalities, see section 5.6.1.

3 **explAIner —** Explainable and Interactive Machine Learning

In this chapter, we propose a framework for interactive and explainable machine learning that enables users to (1) understand machine learning models, (2) diagnose model limitations using different explainable AI methods, and (3) refine and optimize the models. Our framework combines an iterative XAI pipeline with eight global monitoring and steering mechanisms, including quality monitoring, provenance tracking, model comparison, and trust building. To operationalize the framework, we present explAIner, a visual analytics system for interactive and explainable machine learning that instantiates all phases of the suggested pipeline within the commonly used TensorBoard environment. We performed a user study with nine participants across different expertise levels to examine their perception of our workflow and to collect suggestions to fill the gap between our system and framework. The evaluation confirms that our tightly integrated system leads to an informed machine-learning process while disclosing opportunities for further extensions.

The chapter is based on the following publication. For a detailed contribution clarification, refer to section 1.4.

[1] T. Spinner, U. Schlegel, H. Schafer, and M. El-Assady. "explAIner: A Visual Analytics Framework for Interactive and Explainable Machine Learning". In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 1064–1074.

3.1 Introduction

Since the first presentation of neural networks in the 1940s [174], we have seen a great increase in works on artificial intelligence and machine learning. Especially within the last decade, computational resources have become cheaper and more accessible. This development has led to new state-of-the-art solutions, e.g., deep learning, while the increasing availability of tools and libraries has led to a democratization of machine learning methods in a variety of domains [54]. For example, deep learning outperforms traditional algorithms for image processing [175] or natural language processing [176] and can often be applied by domain experts without prior ML expertise [177].

Despite the significant improvement in performance, deep learning models create novel challenges, due to their nature of being *black boxes* [178]. For model developers, missing transparency in the decision-making of deep learning models often leads to a time-consuming trial and error process [179]. Additionally, whenever such decisions concern end-user applications, e.g., self-driving cars [16], trust is essential. In critical domains, this trust has to be substantiated either by reliable and unbiased decision outcomes, or convincing rationalization and justifications [180]. The growing prevalence of AI in security-critical domains leads to an ever-increasing demand for explainable and reproducible results.

Several solutions address the problem of missing transparency in black-box models, often referred to as e**X**plainable **A**rtificial **I**ntelligence (XAI) [36]. Even though AI algorithms often cannot be directly explained [37], XAI methods aim to provide human-readable, as well as interpretable explanations of the decisions taken by such algorithms. XAI is further driven by newly introduced regulations, such as the *European General Data Protection Regulation* [181], demanding accessible justifications for automated, consumer-facing decisions, prompting businesses to seek reliable XAI solutions. A natural way to obtain human interpretable explanations is through visualizations.

More recent work focuses not only on visual design but also on interactive, mixed-initiative workflows, as provided by visual analytics systems [52]. Also, an exploratory workflow [51] enables a more targeted analysis and design of ML models. Visual analytics further helps in bridging the gap between user knowledge and the insights XAI methods can provide. As AI is affecting a broader range of user groups, ranging from everyday users to model developers, the differing levels of background knowledge in these user groups bring along varying requirements for the explainability.

There has been extensive theoretical work on the role of visual analytics in deep learning [54], as well as the synergetic effects this combination can generate [52]. The fields of *interactive* [85], *interpretable* [84], as well as *explainable* [37] ML are also well-studied. While these works bring up a variety of best-practices and theoretical descriptions, they often lack a tight integration into a practical framework. In this chapter, we propose a visual analytics framework for interactive and explainable ML that combines the essential aspects of previous research. Our work is designed to target three user groups. Primarily, we focus on *model users* and *model developers*, as outlined by Hohman et al. [54]. These two user groups are familiar with using and/or developing ML models and are, hence, interested in understanding, diagnosing, as well as refining such models in a given application context [58]. Our third user group, however, are *model novices*. These are non-experts in ML, interested in understanding ML concepts and getting to know more about applying ML models, e.g., for specific domains. Such an educational use of our framework is facilitated through user guidance and interaction monitoring. End-consumers of AI products are not considered separately. Our *XAI framework* is built

upon an *XAI pipeline* that is designed to enable the iterative process of model understanding, diagnosis, and refinement. In addition, to support these three tasks, *global monitoring and steering mechanisms* (subsection 3.2.2) assist the overall explanation process. Figure 3.1 depicts a close-up view of an *explainer*, the main building-block of the pipeline.

In recent research, a variety of concrete XAI methods and implementations have been proposed. However, these tools often are implemented as standalone prototype solutions, lacking an integration into the active ML developing and debugging process. Therefore, a large gap between theory and practice has arisen. As confirmed by our study, most people who are involved in the model usage and development process are familiar with the general concepts of XAI, but most do not have extensive hands-on experience using such tools. Therefore, in contrast to previous work, we not only want to describe the theoretical workflow but use the framework to *operationalize* these concepts in a system implementation, called *explAIner*. We integrate our system in TensorBoard[1], since it is an established tool when it comes to the analysis of DL models. Our system provides an interactive exploration of the model graph, on-demand aggregation, and visualization of performance metrics as well as an integration of high-level explainers such as *LIME* [20] or *LRP* [23]. Based on our framework, our system follows the XAI pipeline and integrates parts of the proposed global monitoring components, such as user guidance.

Finally, we evaluate the implemented system in a qualitative user study with nine participants, ranging from model novices to –developers. During pair analytics sessions [182], we analyze the usefulness of our tool while deriving ideas for future versions.

Summarizing, the main contributions of this chapter are:

1) Conceptual Framework — We propose a conceptual visual analytics framework describing a generalizable workflow for interactive and explainable machine learning.

2) System Implementation — We present the explAIner system implementation, a real-world implementation based on the proposed framework that integrates into the TensorBoard ecosystem.

3) User Study — Finally, we evaluate our approach in a user study with participants across different expertise levels to assess the quality of our approach and its influence on their workflow.

[1] `https://www.tensorflow.org/tensorboard`

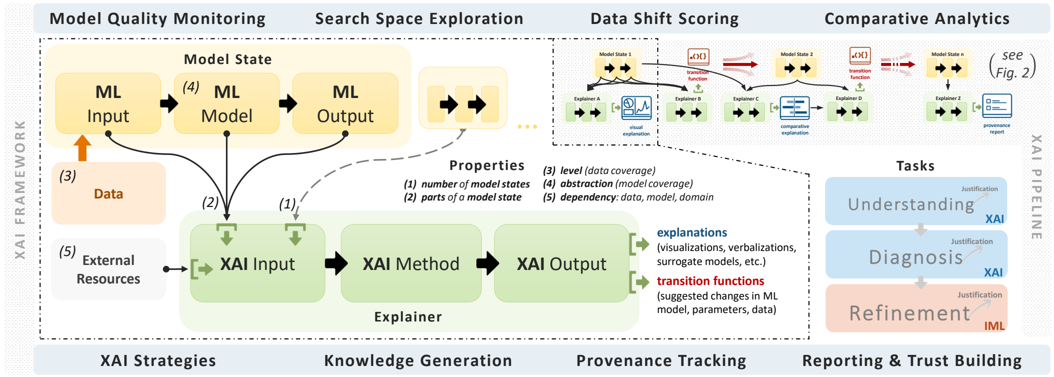


Figure 3.1: Schematic view of an explainer, the key component in our iterative XAI pipeline, which enables the understanding, diagnosis, and refinement of machine learning models. Explainers can be described by five properties, such as the data level they operate on or in which abstraction they explain the model. An explainer accepts one or several model states as input and utilizes an XAI technique to produce either an explanation or a transition function. The pipeline is extended into the XAI framework through global monitoring and steering mechanisms, which help to guide, direct, and report throughout the IML workflow [1].

3.2 Conceptual Framework for Interactive and Explainable Machine Learning

The rise of deep learning has sparked efforts to formalize interactive and explainable machine learning, mostly grounded in theoretical research. However, to effectively develop interactive and explainable systems, a conceptual model must consider practical implementation needs and align with existing theoretical frameworks. Thus, we introduce a conceptual framework designed to enhance the practical application of interactive and explainable machine learning, focusing on practicality and comprehensiveness. The proposed framework formalizes the interactive machine learning and explainability workflow and provides a general foundation for the development of systems for interactive and explainable machine learning.

At the core of our framework is an *XAI pipeline*, an expanded view of the iterative process of developing and refining machine learning models. The pipeline employs explainer modules, which interact with the models to generate *explanations* via visualizations, verbalizations, or surrogate models. The explanations lead to a better understanding of the model, which in turn enables the diagnosis of model flaws and suggest potential refinement strategies. The refinements between model states are described by *transition functions*. Complementing the XAI pipeline are tools for globally tracking and steering the model explanation and development process. Examples of such tools include user guidance, provenance tracking, or reporting. In the following section, we will delve into the details of the proposed framework, starting with the XAI pipeline and then discussing the *global monitoring and steering mechanisms* complementing it.

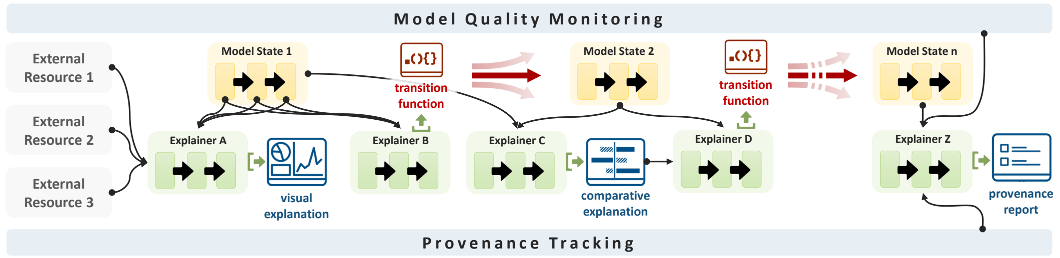


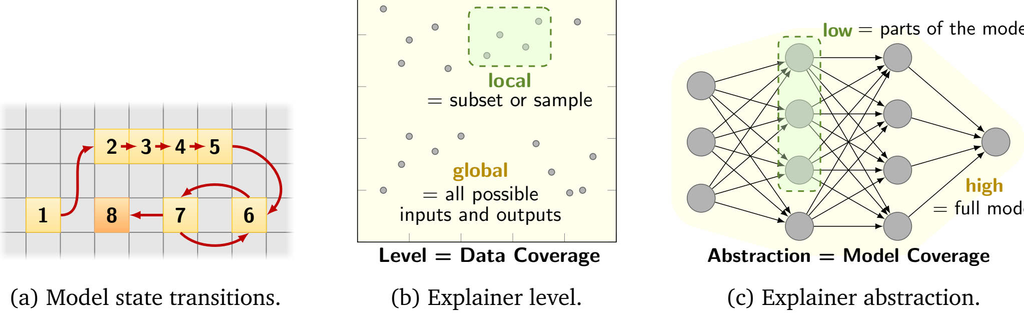
Figure 3.2: Expanded view of an exemplary model explanation and refinement workflow. Explainers are applied to model states to gain insights and pinpoint shortcomings. Based on the identified problems, enhancements are suggested, leading to a transition to a subsequent model state, and so on. Global monitoring and steering mechanisms are employed to record the lineage of the XAI process and to monitor various quality metrics [1].

3.2.1 XAI Pipeline

Summary **Sec. 3.2.1**	We propose an XAI pipeline to structure the IML process, incorporating model states, explainers, and transitions between model states. Explainers take model states as input and output explanations or transition functions, which are used to refine the model, leading to a new, improved model state. Explainers vary in their properties, such as their inputs, data coverage, and model coverage.

Our workflow incorporates multiple *model states* and a variety of explainers, as shown in Figure 3.2. A model state represents a specific configuration of a trained machine learning (ML) model, characterized by its architecture, parameters, and weights. When altering one of these properties, the model transitions to a new state. To identify the most effective model for a particular dataset and task, the space of all possible model states has to be considered. In this context, the aim of interactive machine learning is to facilitate the model's evolution towards states better suited for the problem at hand. More technically, we define any operation that modifies a model state's characteristic as a transition function between two model states in the search space. The process of model refinement, therefore, can be viewed as moving through the space of model states to find the state which maximizes the model's task performance.

Explainers are specialized components that output an explanation or suggest a transition to an improved model state. They consider input from either a single model state (*single-model explainer*) or multiple model states (*multi-model explainer*). In addition to the number of model states, explainers can be characterized by the specific *parts of the model state* they use as input (inputs, outputs, and/or the model itself). Moreover, the *level* of an explainer describes the scope of the data considered, with *global* explainers considering all possible inputs and outputs and *local* explainers focusing on a subset of the data. Complementary, the *abstraction* of an explainer describes the model coverage, with *low* abstraction explainers focusing on a part of the model and *high* abstraction



(a) Model state transitions. (b) Explainer level. (c) Explainer abstraction.

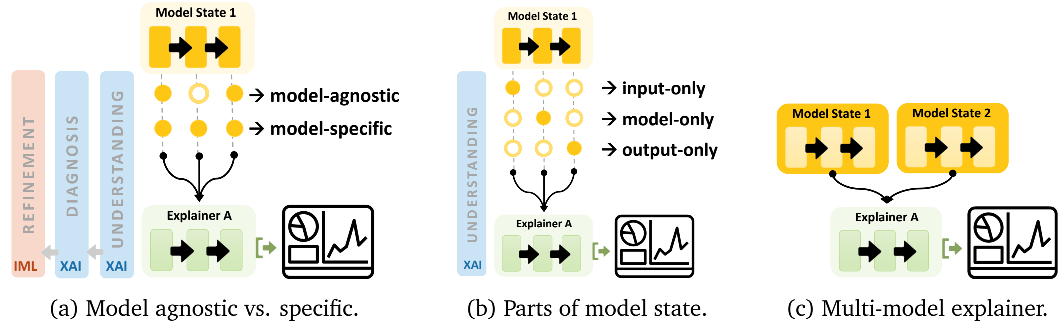
Figure 3.3: Transitions between model states can be described by *transition functions* (a). Explainers can be categorized based on the *level* on which they explain in– and output data (b) and the *abstraction* in which they explain the model (c) [1].

explainers considering the entire model or even multiple models. Finally, explainers can pose different constraints to data, model, or required domain knowledge, which are referred to as *dependencies*.

In the following, we will categorize and describe different types of explainers, divided into single-model and multi-model explainers.

Single-Model Explainers — The first type of single-model explainers are *model-specific* explainers, focusing on the inputs, outputs, and internal mechanisms of a model, making them particularly beneficial for developers who need to examine and refine a model's internal structure or parameters. Such model-specific explainers aid in understanding the relationship between inputs and outputs within a specific architecture. LRP [23] is an example of such an explainer. Complementary, *model-agnostic* explainers work at the data level, treating the model as an opaque entity that transforms inputs into outputs. These are more suited for beginners or users of machine learning models who are less concerned with the model's intricacies and more with its application. Examples of model-agnostic explainers are LIME [20] and Anchors [21]. Figure 3.4a highlights the difference between model-agnostic and model-specific explainers.

Notably, our XAI pipeline does not distinguish between the training and testing phases of a model; it can be applied at any point in the training–testing workflow by capturing the current state of the model, e.g., by writing out a checkpoint. However, depending on the user group, interest might focus more on the progress of the training process or on examining the trained model. Beyond the types mentioned above, we also emphasize explainers that focus solely on either ML input, model, or output, i.e., without considering the other two. This includes statistical analysis of the input data to identify potential biases or explainers only focussing on the model's computational graph to understand its architecture, which is particularly interesting for educational purposes. For example, our implementation includes a "look-up" explainer that offers wiki-style entries to



(a) Model agnostic vs. specific.
(b) Parts of model state.
(c) Multi-model explainer.

Figure 3.4: Different properties of explainers. An explainer can be model-agnostic or model-specific (a), take into account different parts of the model state (b), and focus on a single model state or multiple model states (c) [1].

understand neural network components. Other examples include visualizations of dataflow graphs [35] or node-link visualizations [34].

Multi-Model Explainers — Contrasting single-model explainers, multi-model ones are primarily designed for comparing different model states. They take multiple model states as input, producing outputs like comparative visualizations, model selection tools, or transition functions. Thus, these explainers facilitate exploring the model state search space, allowing for a detailed comparison of various model states. Restricting the analysis to a subset of the search space can help reduce the complexity of the search space exploration. Examples for multi-model explainers include Progressive Learning [74], DeepCompare [73], and Manifold [65].

Furthermore, other types of explainers consider different combinations of inputs, including *external resources*. Figure 3.2 shows how the different explainer types would be employed in an exemplary IML pipeline. Explainer A takes three external resources and model state 1 as inputs, yielding a visual explanation for model understanding or diagnosis. Following this, explainer B focuses on model refinement, proposing a transition function to a new model state 2. Model state 2 is then used as input for the multi-model explainer C, which performs a comparative analysis to the initial model state 1. The comparative explanation, together with model state 2, are used as inputs to explainer D, which suggests further improvements to the model. At the end of the IML workflow, explainer Z summarizes the information collected by the global steering mechanisms *model quality monitoring* and *provenance tracking* to generate a report.

Our pipeline is tailored to different tasks and user groups:

Model novices use it as an educational tool, focusing on understanding and diagnosis to learn about the model architecture and the purpose of its building blocks.

Model users leverage it for diagnosis and reporting, to track and justify their decisions while exploring the model.

Model developers concentrate on the diagnosis-to-refinement loop, using explainers as quality indicators during their IML workflow.

3.2.2 Global Monitoring and Steering Mechanisms

Summary
Sec. 3.2.2

We propose global monitoring and steering mechanisms to guide, steer, and track the XAI pipeline. These mechanisms ensure that users are supported in their goals during all phases of the pipeline. We describe the most important mechanisms, categorized into eight groups.

To effectively manage and track the XAI pipeline, we suggest various overarching mechanisms inspired by related work, grouped into the eight major categories outlined below. Besides the mechanisms mentioned here, our framework can be extended to include additional instruments to adapt to specific tasks and needs.

Model Quality Monitoring — Tracking internal performance indicators like accuracy, precision and recall [183], bias [184], or uncertainty [185] helps pinpoint areas for model improvement. Therefore, we suggest a global monitoring component for the ongoing assessment of the model's quality at its different states.

Data Shift Scoring — In parallel with monitoring model quality, it's important to observe changes in data sources, particularly when the data is dynamic [186]. Monitoring for data shifts is crucial for determining optimal points for stopping training or initiating retraining [187], ensuring the model remains fit for the evolving data.

Search Space Exploration — Efficiently exploring the model's input and output spaces is critical for targeted refinement and optimization. This can be achieved through approaches like Speculative Execution [188], where possible optimizations are explored and presented to the user for consideration before actual implementation. Approaches like automated hyperparameter search (AutoML) [98, 93] allow for the automated exploration of the model's search space.

Comparative Analytics — A vital function is the comparison and selection of different model states. By designing explainers that facilitate this comparison on multiple levels, tasks like model selection [189] and model recommendation [190] become more streamlined.

XAI Strategies — To provide comprehensive explanations, we suggest overarching *XAI strategies* [6]. These strategies can facilitate user guidance [191] or incorporate diverse methods of explanation, such as verbalizations [192]. The role of the proposed XAI strategies is to structure and frame the explanation process within a

broader context. In this framework, explainers are fundamental components that utilize specific strategies and mediums to convey explanations. These components can be arranged in either *linear* or *iterative pathways*, depending on the complexity and nature of the information being explained. Implementing educational strategies, such as grouping several explanation blocks into a single phase, followed by a verification block, is a key aspect of this approach. An important part of implementing these strategies is tailoring them to the needs of different user groups. This involves determining the appropriate level of user guidance required for each group ensuring that the explanations are both effective and user-centric.

Provenance Tracking — During model development and refinement, the *forking paths problem* [193] can arise, where the user is faced with a multitude of options and decisions, possibly leading to false positives. Tracking the temporal evolution of the user's workflow can help mitigate this problem and facilitate the traceability of the user's decision-making process. Therefore, we propose to include provenance tracking mechanisms, such as an interaction tree [194], to track and visually represent the sequence of decisions made by users within the XAI pipeline.

Reporting & Trust Building — Creating reporting components is essential for providing a reasoned justification of the user's decision-making process and effectively communicating the workflow results. In educational or training scenarios, designing these components with storytelling [195] elements can be particularly effective. Besides reporting to other users, these mechanisms can also be used to calibrate the level of trust between user and model [196].

Knowledge Generation — The overarching goal of visual analytics frameworks like the one proposed in this work is knowledge generation. This entails both users learning about machine learning models as well as models learning from user interactions [197].

3.2.3 Use Cases

We present one use case for each user type to make the framework and its application more concrete. Each user is described when solving a typical task and will thus focus on specific elements of our framework.

Case 1 — A computer-science freshman (model novice) takes a lecture on machine learning. As an assignment, the professor provides a neural network model which the students should explore concerning its architecture and functionality. The framework supports the student during the entire task. For example, a single-model explainer, which supplies model-only explanations, could provide information about the model's

architecture in the understanding task. Using provenance tracking, the student can document his process of exploration and summarize it later in the reporting step.

Case 2 — Biologists (model users) want to track the movement of various animals in a zoo. They have to choose between different off-the-shelf models to identify the animals in the images taken by cameras. The proposed framework helps the biologists to decide, which of the models solve the task the most accurate and reliable. Based on a labeled test dataset, the model can be diagnosed using different explainers. Thus possible explainers could be single-model explainers, which deliver model-agnostic explanations to solve the task of verification. Findings then can be directly summarized and reported to the director, justifying the decision for a specific model.

Case 3 — A researcher in the field of self-driving cars (model developer) has built a model which reaches an accuracy of over 99% but always fails in specific situations. The proposed framework supports the researcher in each step of the iterative model development and optimization process. By applying different explainers to his model, he finds that his model always fails when birds are visible in the sky. During refinement, the proposed framework proposes multiple options for state transitions by varying architecture and parameters of the model. Using comparative analytics, the researcher can compare multiple model states based on quality metrics and applied explainers. By iterating diagnosis and refinement, the researcher reaches an accuracy of 99.99%, while at the same time, he can build trust that the model is focussing on relevant parts of the cars surrounding. Since the user wants to advance research in his field, he decides to export and share a report of his model building and explanation process.

3.3 The explAIner System for Interactive and Explainable Machine Learning

Though there already exist systems including some parts of the proposed conceptual framework in section 3.2, we operationalize the framework as a TensorBoard plugin called *explAIner*[2]. The plugin implementation can be seen as an instantiation of the conceptual framework, translating the theoretical concepts to an actual application. We chose TensorBoard as the platform because it is widely used in the ML community, and our system perfectly complements and extends the native functionality it provides. More specifically, we add graph views to augment the graph entities with additional information and allow them to contain actual values and time series, which can be interactively accessed by selecting the nodes. Furthermore, we introduce a global provenance tracking component which allows to store and compare model states persistently. Finally, our system allows the execution of different XAI methods at run-time. By design, XAI methods target specific application domains, data types, or network architectures. We address this heterogeneity by embedding explainers in the proposed VA system, which allows us to react to such constraints dynamically based on the user's needs, e.g., by showing only relevant information (filter) or proposing distinct methods over others (user guidance).

Design decisions for our implementation are primarily guided by the proposed theoretical framework as well as TensorBoards best-practices and capabilities. By splitting the stages of the XAI pipeline into distinct TensorBoard plugins, we aim to ensure separation of concerns [198]. Regarding user interface (UI) elements, TensorBoard gave us an excellent starting point by providing reusable color scales and web components. Furthermore, we try to stick with TensorBoards design habits, such as showing visualizations in overlaying cards (Figure 3.7), maintaining the toolbar layout (Figure 3.6), or keeping

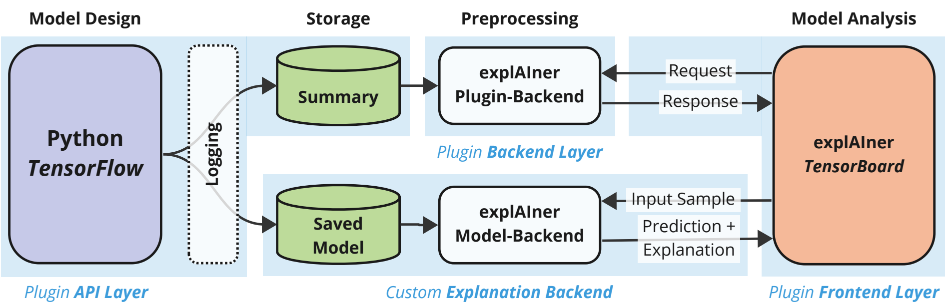


Figure 3.5: Design overview of the explAIner system. The TensorFlow model is created in Python. During training, logfiles are written using the explAIner summary method, which saves graph definition, tensor contents, and the model itself. The explAIner TensorBoard plugin queries data either from the native backend implementation or from an external server, depending on whether the explainer uses tensor data or the model.

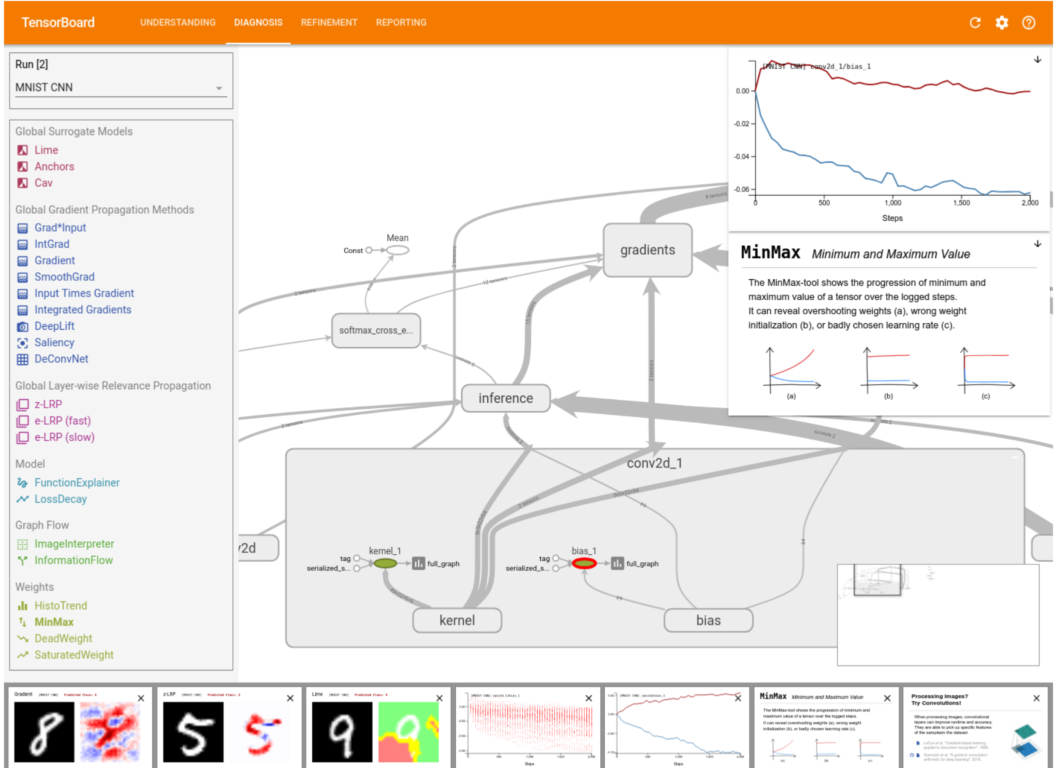


Figure 3.6: The *diagnosis* dashboard. Explainers are arranged in a toolbox-like interface, ordered descending, from high-abstraction to low-abstraction. The graph visualization provides an overview of the full model and allows for node selection. Explanations are shown in the upper toolcard, while information about the explainer is displayed beneath. The provenance bar contains cards from interesting findings.

things contained in specific tabs. Provenance tracking is an exception: TensorBoard is not designed to have components and data shared over multiple plugin tabs, so we have to add this functionality to the TensorBoard system manually.

Figure 3.5 shows the system architecture and its components. The design and training of the TensorFlow model are done in Python manually or by an AutoML or network architecture search approach. We provide an additional explAIner summary, which can save graph definitions, tensors, and the model itself. We store values for each tensor in the graph. Since the required aggregations for our explainers are known beforehand, we can transfer the aggregation step directly to the time of logging. The size of stored data then is comparable to the summaries that are typically written using TensorFlow. Therefore, explAIner has no significant impact on TensorBoards performance.

The explAIner plugin makes use of two different backends, depending on the explanation method. Explainers which work on a model's inputs and outputs use an external model-backend, while explanations for graph tensors use the native TensorBoard plugin backend.

The TensorBoard developers provide an example plugin [199] as a reference for custom plugins. A TensorBoard plugin consists of three parts which can be seen as a pipeline: The **API layer** defines operations to log data during model execution. It corresponds to *model design* and *logging* in Figure 3.5. The **backend layer** loads, preprocesses, and serves the stored data. In Figure 3.5 it handles loading from *storage* and *aggregation*. The **frontend layer** queries data from the backend and renders visualizations in the UI. In Figure 3.5 this is depicted as *request/result* and *model analysis*. These three layers have to be implemented to create a custom plugin. Using the logging operations in the API layer, we extract all relevant data from the computational graph; storage is handled by TensorFlows summary mechanism. Since TensorFlow does not provide a way to save a model as a summary, we complement the API operations by saving the model manually. To execute the model with data, we have to bypass the automated TensorBoard backend layer. In the frontend layer, we can query both backends with similar calls. The plugin can be injected into the TensorBoard UI by providing a custom HTML-page, which, besides the default plugins, loads our custom plugins.

We extend TensorBoard by four additional dashboard-views, one for each step of the XAI pipeline as well as one for global monitoring (reporting). The interface and interaction possibilities for each view follow the specific tasks:

Understanding provides a graph view, enabling interaction with the model to get educational information about its architecture.

Diagnosis builds around an instanced graph view of the model, where variable nodes contain a history of their data at the logging points.

Refinement shows interaction recommendations based on model architecture, findings from previous stages, and general heuristics.

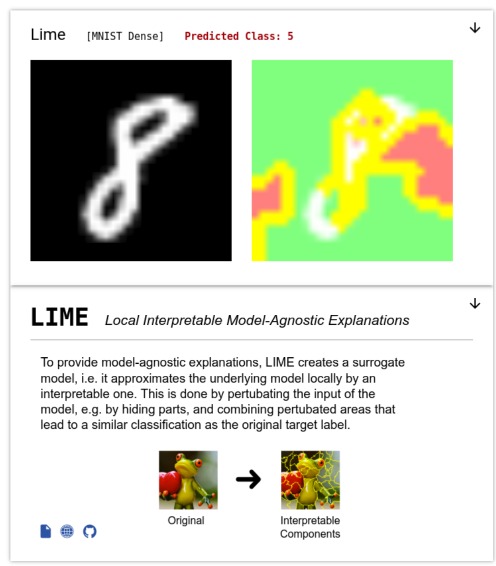
Reporting provides a summary of the full model analysis process. Notes on results can be arranged, annotated, and exported.

To keep track of the knowledge and insights generated during the complete explanation process, our system extends TensorBoard with an additional *provenance bar*. It acts as a persistent clipboard and notetaking-environment, in which the user can document discoveries, thoughts, and interpretations as small provenance cards.

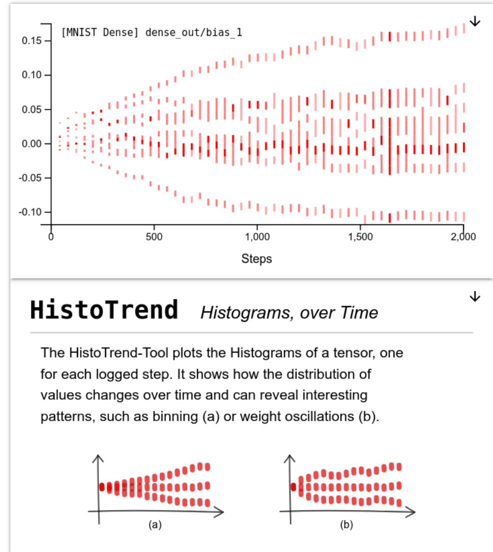
3.3.1 Understanding

The understanding phase is the entry point into our proposed workflow. For a model developer, this step offers information necessary to create a fitting model, e.g., layer sizes, loss function, used optimizer, etc. For a model user and a model novice, it explains

[2] System is publicly available under: *http://explainer.ai/*



(a) LIME (high-abstraction explainer)



(b) HistoTrend (low-abstraction explainer)

Figure 3.7: Information cards showing results for different explainers (top) with the corresponding descriptions for the explainer itself (bottom).

a given model and its functionality by providing visual representations, descriptions and external information on the network. In our prototype system, we implement this phase as the integration of information cards, that can be displayed by interactively focusing parts of a graph representation of the model. While other layouts were considered [200, 102], our graph view is derived from the TensorBoard graph, since it is well-known and reproduces TensorFlow's computational graph accurately. When hovering a graph node, a short description and explaining graphics are displayed. Clicking on a node opens an overlay, containing detailed information and external references, similar to a short wiki article. The content is manually extracted and visually appealingly prepared from wikis, blogs, and scientific publications. Supplementary information can be retrieved for entities of different levels, ranging from the full model down to single operations.

3.3.2 Diagnosis

In the framework, we define the diagnosis phase as the most critical part of our workflow. It enables model novices to visually explore and thus learn about the output of the model. It offers a decision support tool for model developers, that helps them choose necessary refinements, and it gives visual feedback for verification by a model user or domain expert. In our prototype, we offer various explainers sorted by scope, which can be interactively placed on the visual graph representation of the model. We display the visual feedback of the explainers as overlaying cards, which can be saved to the provenance bar to trace the process of exploration. In addition to the explainers output,

we provide a second card with supplementary information on the functionality of the explainer and how its outputs can be interpreted. Figure 3.6 shows a screenshot of the diagnosis dashboard view.

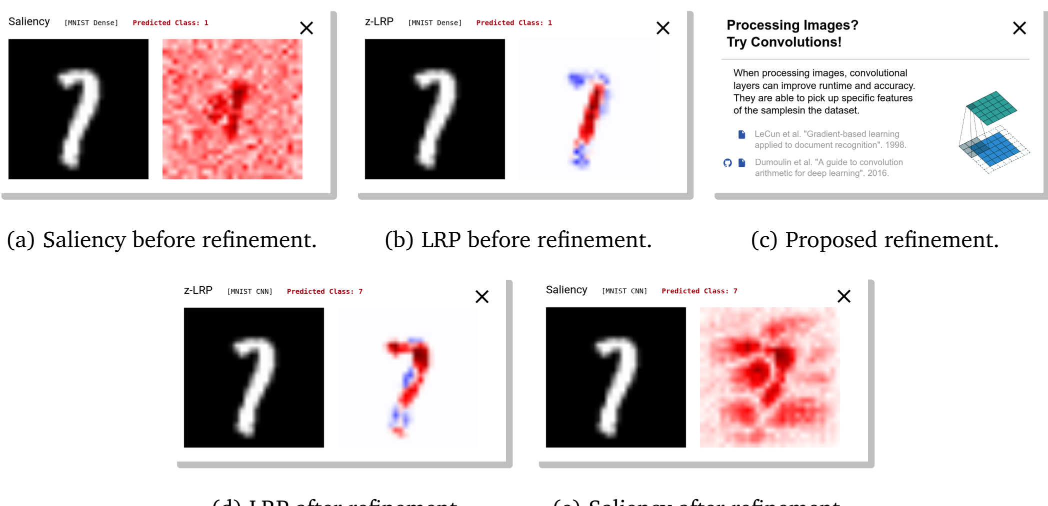
High-abstraction explainers take the trained model and a user-selected sample as input, for which they return prediction and explanation. Low-abstraction explainers work on parts of the model and can be applied to single graph entities or a particular subset of the graph. The explanations range from time-dependent metrics of a single variable up to the visualizations of the flow of a tensor as it traverses the graph. See Table 2.1 for an overview of the explainers we implemented. Since for more advanced networks the graph representation can become quite complex, we provide user guidance to help the user focus on interesting graph entities. This is done by visually emphasizing nodes on which a certain explainer can be applied or by marking nodes that deviate significantly from other nodes of the same type.

Example For a High-Abstraction Explanation – When a user issues a high-abstraction explanation method, e.g., LIME [20], the user is prompted to select a data sample, which is sent to the model backend as input for the explanation. The backend loads the trained model from a saved file and executes the explanation method for the given input. After execution has completed, explanation and prediction are sent back to the explAIner frontend, where they are presented to the user (Figure 3.7a). Besides LIME as an example for surrogate models, we implemented several other model-based explainers, including *Layer-Wise Relevance Propagation* (LRP) and several gradient-based methods, such as *Saliency* and *Deeplift* [23, 26, 28].

Example For a Low-Abstraction Explanation – Low-abstraction explanation methods can be directly applied to individual nodes of a graph. After selecting such node, explAIner creates a request containing identifiers for node and explainer and sends it to the backend layer of the TensorBoard plugin. The backend responds with the aggregated tensor data for the selected node and explanation. Visualization of this data happens directly in the frontend layer of the plugin (Figure 3.7b).

3.3.3 Refinement

The refinement phase is crucial for model developers that want to improve their model interactively. For model novices and model users, this steps is more rarely utilized with the goal of further exploration. In our prototype, we implement two different transitions into the refinement phase. First, the user can choose to add a refinement directly related to a given explainer output. Second, the user can directly enter the refinement phase by choosing the respective tab and choose between all possible refinements. This transition is essential to the model developers since they might already know of more



(a) Saliency before refinement.

(b) LRP before refinement.

(c) Proposed refinement.

(d) LRP after refinement.

(e) Saliency after refinement.

Figure 3.8: Use case showing the provenance bar sequence used for model refinement towards the correct prediction of a seven. A misclassified image is analyzed using (1) Saliency and (2) LRP. As a refinement, explAIner suggests using (3) convolutional layers. After applying the refinement, (4) LRP and (5) Saliency show a correct focus on the relevant features.

general refinements, that are dependent on the context that is given by the explainer. Besides refinements that are targeting improvements of the model accuracy, we also focus on enhancements in space and time requirements of the model. In this prototype, all optimization steps are supported by general textual information to help all users understand their effect. The refinements are realized as recommendations that the model developer might follow to improve its model. Such recommendations give a summary of how the improvement works and why the explAIner system suggests it. Furthermore, improvements that affect the models basic functioning and therefore might change the way a model solves a specific task are provided with links to external resources. This is meant to keep the developer up to date with the latest discoveries in AI since the field develops rapidly. The recommendations that are suggested during the refinement step are based on heuristics, considering graph architecture, the task, that the user seems to be trying to solve, and findings from previous steps.

3.3.4 Provenance Tracking and Reporting

Our TensorBoard implementation is complemented by a *provenance bar*. During the complete exploration and explanation process, the user can save and annotate interesting findings in the provenance bar. While tracking of the exploration process could also be automated, we decided to leave it to the user to directly filter important findings. The provenance bar, therefore, acts as a persistent cross-dashboard as well as cross-model digital blackboard and covers parts of the global monitoring and steering mechanisms

(subsection 3.2.2), namely *provenance tracking* and *reporting & trust building*. Figure 3.8 shows the provenance track for an example model explanation and refinement process.

The reporting phase is the final phase of the frameworks workflow. Its goal is to offer a solution to common issues of missing justification, provenance tracking [195], and reproducibility [201]. In this prototype, we implement the reporting phase as an interactive arrangement of the provenance cards saved in the understanding, diagnosis, and refinement steps. This allows the user to see the feedback by the explainers he acted on, the decisions he made to refine, and, in the case of iterative loops through the workflow, the improved output of the repeated feedback from the explainers. By adding or modifying annotations, the user can document his thoughts and findings and, therefore, structure the process in a storytelling manner. This might be crucial if other people are involved in the model development or deployment process and, hence, justification or trust-building is a necessity. The reporting dashboard accordingly extends the functionality of the provenance bar by the global monitoring and steering mechanisms *comparative analytics*, while further enhancing the *storytelling* and *justification* aspects.

3.4 Evaluation

In this section, we describe the methodology of our study, the feedback we received from the different target users, and the insights we extracted from the given feedback.

3.4.1 User Study

To verify the intuitiveness of our workflow and the usability of the system, we conducted a qualitative user study with different types of target users. We use both a simple and a complex network trained on the MNIST dataset [103], simulating a real-life environment. The goal of the study is to see where the system can be improved and whether all the necessary improvements are already covered in the framework, and, thus, only limitation of this specific system implementation.

Methodology and Study Design – Due to the variety of available interaction loops, we decided to conduct a pair analytics study [182], enabling each participant to transfer their individual workflow to the system. We performed nine approximately one-hour sessions in which a member of our team (henceforth referred to as visual analytics expert) worked with the target user (henceforth referred to as model novice (MN, 🟡), model user (MU, 🔵) and model developer (MD, 🟢)). Each study started with a semi-structured interview regarding the user's previous experience with ML as well as their

expectations on the framework and the system. After gaining these unbiased insights, the visual analytics expert gives a quick introduction to the system, available datasets, and the analysis tasks that the user should solve during the pair analytics session. Then, the control of the system is handed over entirely to the participant. They are asked to communicate their thoughts and actions by "thinking aloud" while conducting the predefined analysis tasks, taking as much time as they need. The visual analytics expert can interrupt the session to clarify interaction possibilities, limitations, or to guide the user towards the next analysis task. The last part of the study consists of another interview reflecting on the difference between the initial expectation and the experience during the pair analytics regarding the workflow, the system, and the performed analysis tasks. All study sessions were audio-recorded and screen-captured.

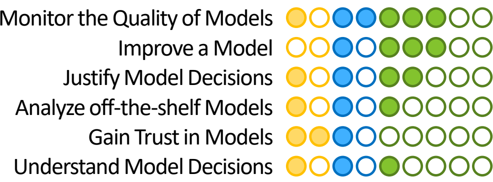
Participants – We selected our participants from three different groups of target users. For the model novices (MN), we interviewed two Ph.D. students with a computer science background that had basic knowledge on ML but had never built NNs before. For the group of model users (MU), we interviewed two Ph.D. students with experience in the analysis of linguistic data but no prior experience with deep learning. For the model developers (MD), we interviewed five experts (two industry developers, three students) that were familiar with TensorFlow and TensorBoard. All participants had either finished or were currently pursuing a university degree. Only one of the participants was female, which could be explained by the low number of females in the domains we were recruiting from.

Tasks – The participants were guided through the interaction along the tasks understand, diagnose, and refine, but were allowed to loop back. In case the participants spent too much time on one task, the visual analytics expert would use unobtrusive questions to guide them to another task.

3.4.2 User Feedback

In the following, we describe the feedback received from participants during the three study phases (expectation, pair analytics, review).In each section, we highlight the aspect that the side-figures address.

Expectations – When we asked participants about the general utility and their expected **use cases for XAI**, the most frequent answers are in line with some of our frameworks global steering and monitoring mechanisms: model quality monitoring, search space exploration, reporting & trust building, and knowledge generation. Additionally, the users from each user group wanted to

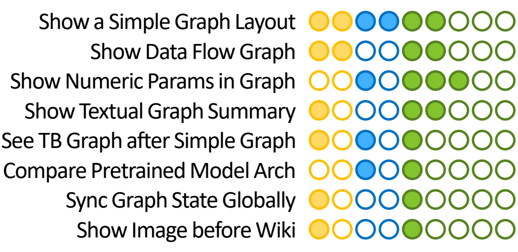


verify pre-trained models with XAI methods, which is also supported by our XAI pipeline. Besides the most frequent suggestions, some users had extraordinary ideas, such as using XAI methods for marketing the model. Within our framework, this could be one manifestation of reporting & trust building. A difference in ideas between user groups is that the MDs had more specific ideas (e.g., feature influence on decision) while the MNs and MUs mostly suggested high-level concepts (e.g., trust building).

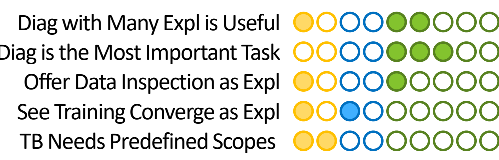
When reviewing the suggested **framework**, most participants agreed with the tasks understanding, diagnosis, and refinement. However, many of them would also prefer to merge the understanding and diagnosis phase. This adaption is supported to some degree by our framework. Depending on the use case, each task of the pipeline can be shortened, left out, or repeated. As we suggest in the description of our framework, the understanding task, for example, is often more important for MNs than for MUs or MDs. Besides the congruent feedback on the framework and pipeline, some individual ideas were presented, such as a separate model building task. While the model building is not a separate element in our pipeline, it can be simulated by starting with a minimal default model and continuous building-block-like refinements.

Pipeline Makes Sense for Eng Task
Merge Understanding & Diag

Pair Analytics – During the **understanding phase**, the users were first presented with a graph representation of their model. The graph representation was criticized by many participants. This is one of the design decisions based on the integration into the TensorBoard environment and does not directly reflect on the framework. Furthermore, it suggests that, in parallel to the explainer toolbox, the graph represents another type of model content for which a toolbox of explanations (e.g., dataflow, classical layout, numeric parameters, text) should be offered. Some individual MDs even suggested showing the code as a representation and not needing the graph, if the model is self-built. Another difference that we see between different user groups is that the MDs focused more on details of the graph such as numeric parameters and code snippets than the MUs and MNs.



During the **diagnosis phase**, participants generally gave positive feedback. In addition to the provided explainers, users wanted to gain insight into the underlying data and other metrics of the model, such as convergence. Such features are instances of the different frameworks explainer types but have not been implemented in our system. For example, a model-agnostic explainer could review the dataset balance and show it to the user. Some further concerns were only



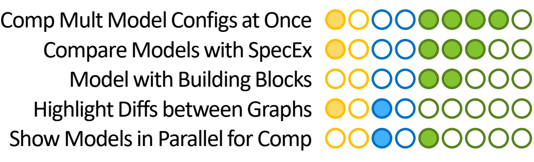
affecting the implemented system, such as scalability and scopes for the calculation of an explainer output. A difference we see between user groups is that MDs value the diagnosis more than MUs and MNs.

Regarding the **toolbox set of** different low- and high-abstraction **explainers**, the most interesting insight was that trust in the explainer's output is an issue for the users. This aspect can, to some degree, be counteracted by giving more guidance within the system. In the framework, this aspect is part of the targeted user guidance within the XAI strategies. In addition to the desired guidance, the participants wanted additional explainers, such as standard metrics and more low-abstraction explainers. Such additions can easily be made in future iterations of the system. A difference we see between user groups is that MUs were less concerned with trusting the explainer output than MDs and MNs.

Use Metrics as Expl in Diag
Do not Trust Explainer Results
Gain Insights into how Expls Work
Reference the Origin of Explainers
Show Metrics during Refinement
Explain Multiple Samples at Once
Explain a Sample/Class at Once
Use Glob Expl on Interm Layers

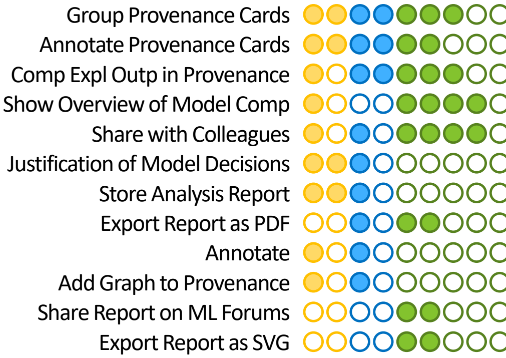
During the **refinement phase**, the feedback and expectations were mixed. Some participants were very optimistic, suggested additional ways to interactively refine the model. All of the suggested refinements (e.g., adding data, changing parameters, switching architecture) are covered by the framework in the form of transition functions resulting from a single- or multi-model explainer. More doubtful participants did not criticize the utility of such a tool, but rather the possibility of offering this functionality with sufficient proficiency. In the future, the current systems should be extended with the suggested refinement methods and additional guidance to select the appropriate refinements.

Concerning the **model comparison**, many of the suggested improvements are in line with the global monitoring and steering mechanisms of the framework, such as search space exploration, data shift scoring, and comparative analytics. They are an important part of future iterations of our system. Interesting individual ideas for future work were a building block system that could adapt the model architecture, the data and the features on the fly and compare different explainers/metrics on a selection of these models. This suggestion in line with a previous suggestion of having a separate model building task in the pipeline and can be simulated with an interactive interface for fast iterations of the refinement phase.



Concerning the overarching aspect of **provenance tracking and result reporting**, the feedback was very unified. All participants liked the feature and would use it to commu-

nicate their results. The importance and acceptance of this feature further confirms the utility of the global monitoring and steering mechanism provenance tracking and reporting & trust building. Within the system, users suggested many interactions, that should be added in future iterations, such as annotation tools, export formats, and layouts. Beyond the systems implementation, participants came up with several suggestions for utilizing this feature, such as different reports to colleagues or stakeholders. A difference we see between user groups is that MUs and MNs target justification and MDs exchange between colleagues.



Expectation Review – Regarding the overall usability and value of the system for real use cases, all participants gave very positive feedback. Half of the participants considered the system too complex for beginners. Two participants stated that it would be okay for either model users or model novices. Only one participant explicitly stated it should only be used by developers. The experienced gap could be fixed by extending guiding mechanisms (see subsection 3.5.2), leaving a workflow which can capture the transition in user expertise while its distinct states can still be applied independently for specific tasks. Common feature requests in this direction focus on additional user guidance in the form of tutorials, suggestions, checklists, information with more labels, and example models.

3.5 Discussion

This section reflects on the design and implementation of explAIner and discusses the results of the evaluation. From the users' feedback, we derive a set of take-home messages and discuss limitations and future work.

3.5.1 Take-Home Messages

From the study results presented in section 3.4.2, we derive several take-home messages for the design of future XAI systems.

(EX1) The Need for a Generalizing Framework — Our literature review revealed that the field of explainable artificial intelligence is highly fragmented, with a large number of different explanation methods and approaches. This highlights the need for

a generalizing framework that allows for the integration of a diverse set of explanation methods, making them accessible in the IML workflow. With the presented XAI pipeline, the classification of explanation methods according to their properties, and the proposed monitoring and steering mechanisms, we provide such a framework, which can be used to integrate existing and new explanation methods. With explAIner, we demonstrate the practical feasibility of our conceptual framework by integrating a diverse set of explanation methods and mechanisms into a comprehensive system. The positive feedback from our participants supports the usefulness and usability of our framework and system.

(EX2) Diverse Use Cases and User Needs for XAI — The study highlights that participants see a variety of uses for explainable artificial intelligence, with model quality monitoring and model improvement being the most common. As to be expected, different user groups have different needs. While model developers primarily focus on model assessment and improvement through detailed analysis, such as monitoring training statistics and exploring feature influence, model novices and –users focus more on model understanding and trust building. This finding underlines the importance of our generalizing framework, which allows for the integration of a diverse set of explanation methods. Furthermore, it highlights the need for flexible XAI systems that can be adapted to different user groups and use cases, for example, through a modular design.

(EX3) Challenges with Graph Representations and Diagnostic Tools — During the understanding phase, the graph representation of models was criticized, indicating a need for simplified and focused representations of the model's architectures. In the diagnosis phase, participants expressed a desire for deeper insights into underlying data and model metrics, highlighting a demand for more comprehensive diagnostic tools. In section 4.3.3 of this thesis, we present a set of mechanisms to address these challenges.

(EX4) Breaking Up the Phases of the IML Loop — Our participants' suggestion to merge the understanding and diagnosis phases indicates that the phases of the IML loop are not always performed in a linear fashion and often overlap or happen in parallel. We, therefore, break up the IML loop in the frameworks and systems presented in the following chapters, allowing for a more flexible and iterative workflow.

3.5.2 Limitations and Future Work

Overall, the feedback on the system design was positive. Additionally, the users had several ideas for complementary features, for which integration in the system is planned for future versions. The most requested functionality was a simplified presentation

of the model graph, with an option to switch to the more complex representation if required. Another significant point was trust in the explanation methods themselves. By including additional descriptions, links, and possible interpretations for every explainer, we tried to improve the confidence in the explanation. This idea of *meta-explanations* could be extended even further: instead of providing static descriptive content, dynamic visualizations could be rendered to explain the current explainer output, e.g., the surrogate-models architecture for LIME or a heatmap displaying the relevance in all layers for LRP. Regarding the expert users, more advanced features were desired. Some users wanted to apply high-abstraction explanations on a subset of layers [61]. Most commonly requested was an additional view to enable direct comparative analytics as well as the speculative execution of proposed refinements. The refinement step was often rated to be the one with the highest potential, but it was also considered the most complex to implement. Ideas to enhance its functionality included proposing code fragments, providing building blocks, or even scaling it up to a social platform were suggestions for model improvement could be shared among other developers. This could be extended by a way to restore saved exploration states or reproduce the process of exploration from other users.

By further extending user guidance, for example, by including AI driven recommendations, a broader range of user groups could be addressed. The additional flexibility could make the tool suitable for educational purposes or more advanced analysis and refinement tasks.

While some of the user suggestions can be included in our existing system, e.g., a simplified graph view, others are not that easy to realize, e.g., on-demand refinement. While building upon TensorBoard saved us a significant amount of work for data logging, data loading, and developing the graph view, it also presented some limitations. This concerns the user interface as well as export, storage, processing, and exchange of data. The strict separation of tabs as *plugins*, each with its distinct data backend, makes data sharing as well as mutual views (e.g., provenance bar) challenging to realize. Furthermore, to close the IML loop of model development (TensorFlow) and model analysis (TensorBoard), those stages have to be combined. While we were able to surpass some of these issues, e.g., by including an external backend or modifying the website template, a system which could provide a full IML and XAI workflow would require a more specialized architecture.

Finally, some users asked for additional, in particular low-abstraction, explanation methods. We deliberately designed the proposed framework, as well as the derived system, as a platform for the integration of already existing and entirely new explanation methods.

3.6 Conclusion

In this chapter, we presented explAIner, a framework for interactive and explainable machine learning, capturing the theoretical and practical state-of-the-art in the field. As a core concept of our XAI framework, we defined the XAI pipeline, which maps the XAI process to an iterative workflow of three stages: model understanding, diagnosis, and refinement. By determining additional global monitoring and steering mechanisms, we extended the XAI pipeline by overarching tools and quality metrics. To show the practical relevance of our framework, we instantiated it in an actual system. Besides the three stages of the XAI pipeline, the implementation covers global monitoring and steering mechanisms by providing provenance tracking as well as an additional reporting step. To test the usability and usefulness of our tool, we performed a user study with nine participants, coming from different user groups. The users found our system to be intuitive and helpful and considered an integration in their daily workflow.

iNNspector — Visual Deep Model Debugging

4

Deep learning model design, development, and debugging is a process driven by best practices, guidelines, trial-and-error, and the personal experiences of model developers. At multiple stages of this process, performance and internal model data can be logged and made available. However, due to the sheer complexity and scale of this data and process, model developers often resort to evaluating their model performance based on abstract metrics like accuracy and loss. We argue that a structured analysis of data along the model's architecture and at multiple abstraction levels can considerably streamline the debugging process. Such a systematic analysis can further connect the developer's design choices to their impacts on the model behavior, facilitating the understanding, diagnosis, and refinement of deep learning models. Hence, in this chapter, we (1) contribute a conceptual framework structuring the data space of deep learning experiments. Our framework, grounded in literature analysis and requirements interviews, captures design dimensions and proposes mechanisms to make this data explorable and tractable. To operationalize our framework in a ready-to-use application, we (2) present the iNNspector system. iNNspector enables tracking of deep learning experiments and provides interactive visualizations of the data on all levels of abstraction from multiple models to individual neurons. Finally, we (3) evaluate our approach with three real-world use cases and a user study with deep learning developers and data analysts, proving its effectiveness and usability.

The chapter is based on the following publication. For a detailed contribution clarification, refer to section 1.4.

[2] T. Spinner, D. Fürst, and M. El-Assady. "iNNspector: Visual, Interactive Deep Model Debugging". 2024. arXiv: 2407.17998.

4.1 Introduction

Despite the recent prevalence of deep learning algorithms, the model-building process is still a trial-and-error one, primarily guided by the developer's experience or a few known best practices [58]. Typically, different architecture– and hyperparameter configurations are tested manually or by automated architecture search [93], from

which the best-performing model is then selected and further refined. Assessment and selection are often based on performance metrics, such as loss or accuracy, which highly aggregate information and leave the model's inner workings opaque [83]. However, conflicting with deep-learning model's black-box character, a deep understanding of the model's inner workings is essential for well-informed model diagnosis, verification, and refinement. Furthermore, such understanding is fundamental for trust-building and model verification, e.g., for its deployment in safety-critical environments such as in the automotive or medical sector.

Techniques for explainable artificial intelligence [39] try to tackle this challenge; however, many approaches are model-agnostic, leaving the model's inner workings opaque, and operate on a local level, explaining decisions only for single input samples. To achieve a deep understanding of the model and establish a connection between its behavior and the underlying architecture, a more holistic view on the model and its development process is needed. Therefore, we argue that the structured *debugging of neural networks* should become a routine in the daily workflow of model development and –design. Complementary to the debugging in traditional software development Layman et al. [202], we define the debugging of deep learning models as follows:

Definition **Debugging of DL Models**	We define the debugging of deep learning models as the process of identifying and fixing errors in the **model's architecture**, **behavior**, or **training data** either during **training** or **post-training** by iteratively calibrating the understanding of the model behavior through the structured **analysis** of the **model's architecture, hyperparameters, performance metrics, learned features, activations**, and **outputs**.

In contrast to traditional software debugging, there often is no definite connection between an error and its source. Therefore, error identification in deep learning models might involve several techniques used separately or in combination. For instance, **local output analysis** can help identify instances where the model performs poorly or provides incorrect predictions and then determine why these errors occur. In contrast, **global output analysis** can reveal patterns in the model's behavior over the entire input data space, e.g., to identify biases or overfitting. Methods for **interpretability and explainability** provide additional tools to interpret the decision-making process of a model. For instance, attention maps in convolutional neural networks can help understand which parts of the input are most influential in the model's decisions. While the previous techniques focus on in– and output data and its model-internal representations, **architecture analysis** focuses on the model's computational graph. For example, the graph can be analyzed to identify unusual or erroneous combinations of building blocks like layers or operations. Particularly relevant to assess the success of a model are **performance statistics**. Overfitting, for instance, can be detected by checking the learning curve, which plots the model's performance on the training

and validation datasets over training epochs. Finally, **model statistics** use statistical methods to understand the model's internal state, such as weights and activations or output characteristics, for example, to identify model biases. **Visualizations** support the abovementioned techniques, helping the analyst understand the results. E.g., a visual representation of the architecture graph can reveal suspicious patterns [35], or a plot of activations can reveal a neuron's focus [203].

This way of debugging allows the developer a holistic view of the model and, thus, is equally relevant in cases where there are no apparent errors in the model. In cases where the model is not behaving according to the developer's intention, systematic debugging also allows for the identification of error sources, indicating possible areas for model refinement.

Existing commercial and open source systems supporting model explainability and debugging [105, 114] do *generalize* well to different data, models, and tasks. However, they come with narrow limitations regarding the richness of available tools, the data that is accessible after finishing the training stage, and their provided visualizations and interactions, leaving the debugging process shallow. In contrast, there has been a multitude of visual analytics systems for explainable deep learning in the last years [52, 54], allowing for *thorough* investigation of models. They provide highly specialized tools, visualizations, and interactions, with the downside of substantial limitations regarding their transferability to different data, architectures, or tasks. With this work, we argue for the need to combine *generalizability* and *thoroughness* to bridge the prevalent research gap in state-of-the-art techniques for model debugging. Hence, we aim to answer the following research questions arising from the identified research gap: *which steps and data in the model development workflow are of interest to the developer for debugging? Following that, what are the mechanisms, interfaces, and visualizations a system should implement to make these units of analysis accessible?*

To enable this kind of systematic debugging, the machine learning community has to get away from tedious single-item inspection, often requiring the manual setup of logging mechanisms, time-consuming re-training of the model, and single-use data visualization pipelines. Instead, all data arising in a machine learning experiment must be readily available during the debugging stage. Additionally, human-interpretable representations of this data should be effortlessly generated by the machine learning developer, minimizing the entry barrier for systematic model debugging.

To tackle these challenges, in this chapter, we contribute:

1) Debugging Framework — We capture and structure the design space of systematic model debugging, focusing on the entirety of data generated in machine learning experiments and the necessary mechanisms to explore it. We provide an integrated conceptual framework that combines the structured components into an overarching

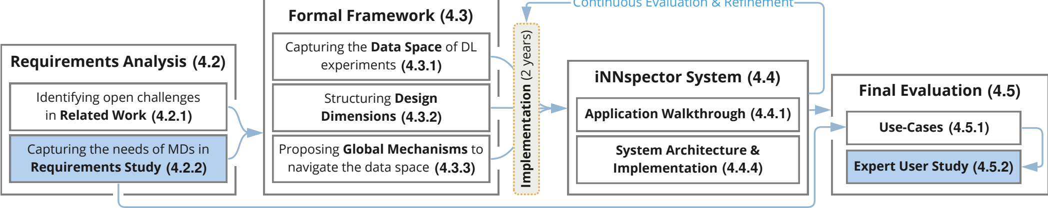


Figure 4.1: The workflow we follow to substantiate, design, implement, and evaluate the iNNspector system for systematic model debugging.

approach. The framework captures the different aspects to be considered and establishes fundamental guidelines for designing real-world model debugging applications. To date, such comprehensive characterization and systematization of the debugging process is missing in the field.

2) System Implementation — With iNNspector, we present a comprehensive system for the systematic debugging of deep learning experiments. The system instantiates the mechanisms and guidelines formalized in our conceptual debugging framework. The system is released as open-source software and is designed as a platform for real-world model debugging scenarios, going beyond the sole demonstration of our conceptual framework's actionability. iNNspector is a new kind of debugging system, providing generalizing access to *all* data arising in the deep learning workflow through implementing appropriate visualizations, navigation patterns, and an extensible tool palette.

3) Requirements Analysis and Evaluation Study — Our conceptual debugging framework and system implementation are grounded in current research gaps, and a qualitative requirements study with real-world model developers. We capture the developers' everyday debugging workflows and –needs, revealing significant challenges in the state-of-the-art. Finally, we evaluate our system with model developers, assessing the applicability and usefulness of our approach and how it closes the identified gaps.

To establish our approach, we follow the workflow of substantiating, designing, implementing, and evaluating the iNNspector system for systematic model debugging, diagramed in figure 4.1. Starting with an extensive requirements analysis (section 4.2), we identify open research challenges from related work (section 4.2.1) and capture the needs of model developers in a requirements study (section 4.2.2). Following, we structure these requirements into a concecptual framework (section 4.3), covering the different aspects that have to be considered for the design of a system supporting systematic model debugging. Namely, this includes the data that is relevant during the debugging stage (section 4.3.1), the design dimensions that have to be considered in such a system (section 4.3.2), and the mechanisms the system needs to implement to make the data explorable by the analyst (section 4.3.3). Based on the conceptual framework, we present the iNNspector system for systematic model debugging (4.4). In a

detailed application walkthrough (section 4.4.1), we explain the interface, visualizations, and interactions the system implements, linking them back to the requirements analysis and the framework. A short description of the system architecture and –implementation (section 4.4.4) explains how iNNspector can be integrated into the existing model development workflow. We evaluate our approach based on use cases (section 4.5.1), showcasing all components of the system. The use cases are the foundation for an expert user study, which evaluates the usability and usefulness of system and framework.

4.2 Capturing the Need For Systematic Model Debugging

To effectively tackle our goal of facilitating and establishing the systematic debugging of deep learning experiments in everyday machine learning workflows, we start by capturing research gaps and open challenges in existing academic and industry works. We ground the design decisions of our approach in requirement interviews with real-world deep learning developers. We gather shortcomings in their existing tool chains, collect real-world use cases where systematic debugging would have been crucial, and capture requirements for a system enabling such debugging.

4.2.1 Research Gap and Open Challenges

Summary
Sec. 4.2.1

There exist multiple research gaps and challenges in the domain of explaining and debugging neural networks. Current **visual analytics** approaches are **highly specialized** either in terms of network types, domains, or goals. While this specialization is necessary for deep insights, **it limits** the techniques' **generalizability and transferability** to broader deep learning workflows. Existing **XAI techniques** mainly offer local explanations and are often model-agnostic. They do **not effectively explain the entire space of model inputs and outputs** and **cannot trace errors back to model architecture**. While **neuron- or layer-wise explanations** can provide such global views, they are limited by their **lack of integrated access to** both **the technique itself** and **lower levels of the network**. **Model tracking** is underrepresented in existing work, lacking visual representations of model iterations and changes between these iterations. **Implementation complexity** affects all of the above: elaborate data provisioning and intricate implementation act as a barrier, hindering the widespread adoption of debugging and explainability techniques. The identified challenges emphasize the need for a general-purpose framework to address these gaps.

Systematic model debugging has recently gained momentum in the machine learning community as the foundation for trust-building, informed decision making, and effective

model refinement [17]. In the following, we identify research gaps in the state-of-the-art, substantiating why this kind of debugging currently is not common practice and what open challenges have to be resolved to close these gaps.

Overspecialization of Current Visual Analytics Techniques — Current research addressing the debugging of machine learning models predominantly focuses on specialized network types, domains, and analysis goals [52, 54, 53].

▷ Example **Specialization of VA in ML**	Most of the recently presented approaches target a highly specific problem; thus, the presented VA techniques are also highly specialized. For example, Kahng et al. [60] propose an approach for investigating and debugging GANs *(specialization on network type)*, Strobelt et al. [63] present a tool for the debugging of sequence-to-sequence models for text translation *(specialization on network– and data type)*, and Cabrera et al. [204] focus on detecting intersectional bias *(specialization on analysis goal)*.

It should be noted that the specialization of these tools is essential to facilitate the deep insights they can provide and to solve the specified tasks, and, therefore, is not a shortcoming. However, our approach aims at a general technique enabling model debugging in the everyday deep learning workflow. In this setting, the specialization of the presented tools limits transferability and generalizability, rendering them inappropriate for our goal. Particularly, with iNNspector, we strive to structure the debugging process in a way that allows for the integration of specialized techniques while still providing a general-purpose framework for systematic model debugging.

Limitations of State-of-the-Art XAI Methods — Prevalent XAI techniques primarily provide local explanations [37], rendering them incapable of providing explanations over the entire space of possible model in- and outputs. Furthermore, the explanations generated by these approaches are often model agnostic, leaving the inner working of the model and its influence on the model behavior opaque.

▷ Example **Limitations of Current XAI Methods**	LIME [20] and LRP [23] are two popular XAI techniques that provide local explanations for a model's predictions, i.e., they explain the model's behavior for a single input sample. The results, often used to annotate the input with a heatmap, can reveal the most relevant features for the prediction. This conveys two significant limitations: First, only known inputs can be inspected. A prominent example is Tesla's autopilot misinterpreting the moon as a traffic light [16], which could only be identified as problematic beforehand by LIME and LRP if images of the moon were part of the test data. Second, the explanations are post-hoc and, therefore, do not provide any information on *why* the model behaves the way it does.

This leaves such XAI techniques limited regarding the systematic model debugging since, typically, the identified errors cannot be projected back onto architectural issues and possible refinements. The narrow constraints of existing XAI implementations in their applicability to specific data types and model architectures further complicate their usage as a general debugging tool.

Missing Access to Neurons — Neuron- or layer-wise explanations, such as feature visualization [112] or attribution graphs [113] are a powerfull global alternative to the data-specific, local explanation of single samples. Here, integrated access to both the techniques themselves and to lower levels of the network seems currently the most limiting factor.

Insufficient Model Tracking — The problem of model tracking is mostly ignored by related work and only covered on a technical level by commercial or open-source tools, relying on tabular representations [114] or Git logs [205]. However, visual representations of model iterations and changes between iterations are currently missing.

Lack of Accessibility — Finally, all of the previously mentioned aspects share the fundamental challenge of requiring elaborate data provisioning and intricate implementation, preventing the widespread use of debugging and explainability techniques in the daily workflow of deep learning developers. Thus, ease of access and adaptability to individual tasks and domains is crucial to establishing the systematic debugging of deep learning models as a common practice.

4.2.2 Capturing Developer Requirements

Summary
Sec. 4.2.2

We interviewed three model developers to identify requirements for a system aiding in deep learning debugging. The results show that: (1) Model management is crucial, especially change detection and model comparison. (2) Developers need access to high-level performance metrics and runtime visualizations. (3) Different levels of architectural detail are desired for comparing or closely analyzing models. (4) User guidance and automated mistake detection are considered useful. (5) Varied interest exists in visualizing neuron activations and network weights.

To capture the needs and expectations towards a system facilitating systematic debugging of deep learning experiments, we conduct structured requirements interviews with three real-world model developers. In each interview, we capture the field of expertise of the developers, including tasks, data types and model architectures they typically deal with, as well as their toolchain and workflow. Subsequently, the interview focuses on the developers' current debugging workflow, covering habits, used tools, and, particulary,

concrete challenges and use cases they experienced in their work where an in-depth debugging was necessary to finally resolve issues with the model. The use cases used for the final evaluation of our approach (see section 4.5.1) are inspired by the experiences and workflows our developers reported in the requirement interviews. As foundation for the design of our system, the interview continues with open-ended questions on expectations towards such a system. Particularly, we ask about the parts of the model the developers are interested in during debugging (e.g., different parts of the architecture or metrics), and how common tasks (e.g., model comparison) could be solved in such a system. Finally, we present early ideas of how such a system could work, particularly, how we plan to use the structural backbone (see section 4.3.3, M2) and navigation over different levels of model abstraction (see section 4.3.3, M3) to provide access to all entities of an experiment.

4.2.2.1 Requirements from the Perspective of the Model Developers

Summarizing the results of our study, we identify the following requirements from the perspective of the model developers. For additional results of the requirements study, including descriptions of the developer's current workflows, the data relevant for debugging, and concrete debugging use cases, please refer to section 4.2.3.

(R1) Model Management — At the top of their iterative model development and refinement workflow, the model developers name model management a crucial task. While this also involves the organization and storage of source code and models, the most highlighted functionalities in this task are change detection and model comparison. All interviewed developers use, besides TensorBoard, customized workflows for model management, often relying on simple tools like filenames, folder structures, and command-line scripts.
(R2) Performance Metrics — For high-level model overview and comparison, the model developers want to assess the performance of models based on the visualization of summarizing metrics, like loss or accuracy. Also, visualizing the model's runtime performance is requested for time-critical environments.
(R3) Architecture Representation — The developers wish for an abstract architecture representation when comparing multiple models. For the closer analysis of one specific model, more architectural details and information about the model's parameterization are requested. When viewing a single model, the flow of data in the model is relevant to the developers, as well as the individual layer capacities and operations.
(R4) Inspection of Weights and Activations — When the analysis gets more detailed, e.g., when searching for specific issues in a model, the developers ask for visualizations of underlying data. Thereby, they seem more interested in the activations of neurons on single samples or whole dataset classes than in the trainable weights of the network.

For both, they identify use cases where they need to inspect the distribution of values to detect issues with bottlenecks, gradients, or network capacity. However, the tasks and requirements involving data inspection and visualization varies strongly between developers. While some prefer a summarizing representation of data with no interest in weights or activations, others describe use cases in their development workflow where the detailed depiction of such data plays a crucial role. Notably, the developers mention that all network parts are relevant for model explainability. For the analysis of weights and activations, an even closer focus on specific layers is wished for. Particular interest is shown in visualizing the development of such variables over time, for example, for convolutional kernels.

(R5) Guidance and Abnormality Detection — User guidance is mentioned as a useful extension by the developers to cover the vast search space that a systematic model analysis opens. Especially the automated detection of careless mistakes, such as accidentally applying an activation function twice after a layer, is a frequent requirement.

4.2.3 Capturing Developer Workflows and –Challenges

Summary
Sec. 4.2.3

Developers in our study report disjointed workflows in deep learning model development and debugging. Their development is usually based on existing architectures and iterative trial-and-error refinement, hindered by a lack of guidelines and difficulties in detecting mistakes. They use various tools for tracking iterations and assessing model performance. Most debugging relies on a trial-and-error approach, with systematic methods rarely used.

In the following, we report additional insights we gained from our expert user study, structured into observations of the current workflows and their implications for model debugging. Overall, we perceive the workflows of our developers as disjointed, with the individual tasks unconnected and a variety of separate tools involved.

4.2.3.1 Current Workflows and Observations

As a baseline, we capture the interviewed developers' conventional model development– and debugging practices, which we summarize in the following.

Current Model Development Workflow — All developers report beginning their workflow by researching promising architectures from related scientific resources (e.g., papers, blogs) and existing approaches (e.g., GitHub repositories). Then, based on best practices and personal experience, they adapt these architectures to create and train an initial architecture for the intended task. Following an iterative trial-and-error approach, this initial architecture is refined and re-trained until the results are satisfying. As a reason

for this fuzzy procedure, the developers mention a lack of guidelines, best practices, and suitable tools for model inspection. Furthermore, hard-to-detect careless mistakes are identified as the most hindering obstacle in the model building and refinement process. To track the different model variants and –iterations arising in this workflow, the developers rely on folder structures and file names, Jupyter notebooks, custom shell scripts, or the output of automated hyperparameter optimizers, c.f., Hyperopt [98]. For model assessment, they reportedly rely on performance metrics (e.g., loss or accuracy), which are logged and inspected using graphical tools (e.g., TensorBoard), Jupyter notebooks, or as raw (textual) command-line outputs.

Current Model Debugging Workflow — To deduce actual needs and requirements for our system's design, we capture our developers' current debugging practice, including possible frustration factors and discontents with their current tool chain. Only one of our participants reports that he "occasionally" systematically debugs his models, usually when there are fundamental problems with the model's performance. When doing so, he would debug in a top-down approach, inspecting questionable model outputs, checking the activations of layers and neurons, and investigating other experiment factors, such as domain-specific loss functions. The other participants would still rely on their trial-and-error-based process, change entirely to a new architecture, or, sometimes, use local XAI methods on individual dataset samples. The quality assessment of a model in the trial-and-error process is mainly based on the local inspection of dataset samples combined with attribution methods. As problems in this process, one of our participants names missing best practices and a lack of leverage points to refine models.

4.2.3.2 Implications for Model Debugging

As the core of our requirements study, we collect wishes and open challenges from our model developers, which they experience in their daily workflows. From these insights, we derive implications and actionable suggestions on how a system enabling the systematic debugging of deep learning models should look like.

Important Data for Model Debugging — The debugging of deep learning models involves a variety of data arising in the machine learning experiment (c.f., section 4.3.1). Therefore, our interviews capture the data that our model developers consider important for model assessment and –comparison. Textual and graphical representations of performance metrics were rated by far the most relevant. For time-critical use cases, this also includes the computational complexity of the network. For high-level model comparison, abstract hyperparameters like the number of layers and number of trainable parameters were named important. For more detailed architecture inspection, layer types, activation functions, loss functions, optimizers, and learning rates were of interest

for our developers. Particularly important for debugging was the possibility to log and visualize activations for data samples.

Use Cases Suggested by Study Participants — We collect a variety of use cases, where either in-depth debugging or the ability to inspect particular data in the model post-training would have been useful.

Careless mistakes were reported to be the most prevalent frustration factor, often affording time-consuming narrowing down of the problem. One developer mentioned the use of a sigmoid activation in the last layer as cause of such error, naturally preventing the model to adapt to a regression task. However, often the root cause of such seemingly simple problems is hard to identify due to the abundance of possible error sources. With easy, simultaneous access to model architecture and the shape of activation distributions, this issue could likely be resolved much quicker. Another developer listed diverse similar situations, including vanishing gradients through the wrong choice of activation functions, erroneous data preprocessing (wrong features boosted, timestamps not converted to UTC), or confused order of Numpy matrix indices. Notably, the latter situations emphasize a fundamental challenge in the debugging of deep learning models: their tremendous ability to adapt to arbitrary data makes the model learn *something*, even if the data is wrongly (pre)processed. In conjunction with the model's opaqueness, issues are hard to detect and locate since no strict exception occurs; instead, the results often just get (slightly) worse.

Aside from these everyday-mistakes, several other use cases for systematic debugging were identified, most common of which was to balance losses (e.g., the generator and discriminator in GANs) or to assess the influence of frequently used techniques (e.g., dropout, activation functions, or batch normalization).

Complementing the expert requirement interviews, we captured additional use cases voiced by data science– and machine learning researchers outside our expert study group. One researcher questioned the usefulness of 1×1 convolutions typically part of inception blocks. Observing the layer in– and output and statistics on the layer activations could be helpful to evaluate their use as a dimensionality reduction technique [206]. Another researcher mentioned beginners mistakes as a significant frustration factor when entering the field of deep learning. Particularly, he got confused when experimenting with auto-encoder tutorials and mistakenly used class labels as optimization target. One machine learning engineer had a particular interest in observing the values of different kernel initializations post-training to evaluate their influence on the formation of capable sub-networks [207]. Finally, one deep learning developer experienced stagnant training and poor reconstruction quality when building a variational auto-encoder. After hours of manual debugging, he realized that a leaky relu activation function in the latent space interfered with the gaussian shape of the latent distribution. This could have been

easily recognized with suitable tools to inspect the distribution of latent variables in combination with a detailed architecture representation.

The Importance of Systematic Debugging — Surprisingly, our participants occasionally show a very conservative opinion about using systematic debugging to substantiate their decision-making in the model building and verification workflow. For example, when asked whether they would like to inspect and attribute the responses of single neurons, e.g., to determine the ideal size of a GAN's latent space, they instead wanted to stick to best practices and personal experiences: "in a GAN, always use 100."

We argue that this example highlights how model inspection and debugging should be a fundamental part of every informed DL workflow; the factors playing a crucial role in the performance of a model should not be chosen based on guesses, leading to serendipitous results and leaving the potential of an architecture unexplored.

4.3 Conceptual Framework for the Systematic Debugging of DL Experiments

Several fundamental requirements and mechanisms have to be considered to facilitate the systematic debugging of deep learning experiments, which we strive to formalize with the conceptual framework presented in this section. The framework provides guidelines and serves as a checklist for designing systems for the debugging of deep learning experiments.

We structure this section by the two substantial aspects that have to be considered for the design of such a system: the various types of data arising in deep learning experiments (section 4.3.1) and the necessary mechanisms and visual representations required to make this data effortlessly accessible by the deep learning developer during debugging (section 4.3.3).

4.3.1 Capturing the Data Space of Deep Learning Experiments

Logging quality metrics and model checkpoints is insufficient for effective model debugging. Thus, we define the **data space** of a machine learning experiment as all data arising during the machine learning workflow and classify this data according to its **origin**, **dimensionality**, and **variability over time** into four main categories: **structural data**, **scalar data**, **high-dimensional data**, and **functions**. This classification builds the foundation of our conceptual framework by formalizing the data that should be made available to the deep learning developer during debugging.

Category	Domain	Sub-category	Variability	Examples
Structural (D1)	$G = (V, E)$	Tree of Models	constant	—
		Architecture Graph	constant	—
Scalar (D2)	$\mathbb{R}$	Hyperparameters	constant	Batch Size
		Metrics	variable	Loss, Accuracy
***n*-dimensional (D3)**	$\mathbb{R}^n$	In– and Output (∼ *Dataset*)	constant	Time-Series (1D) Images (2/3D)
		Weights	variable	Bias (1D) Dense Kernel (2D)
		Activations	variable	—
Function (D4) c.f., Architecture Graph	$\mathbb{R}^n \to \mathbb{R}^n$	Structural	constant	Reshape Concatenate
		Mathematical	constant	Convolution Activation Function

Table 4.1: Overview of the different types of data arising in the deep learning workflow, each having different domains, origins, and instantiations. If the data is changing during training it is categorized as *variable*, otherwise as *constant*.

The assessment of model quality and training progression is usually based on quality metrics logged during the training process. Together with model checkpoints, which encode the model instance at a certain point for later use, they form the entirety of data being typically logged during machine learning experiments. However, we argue that for systematic model debugging, various other data arising in the DL workflow is relevant and should be considered during the analysis phase.

Therefore, extending the *data* term used in the machine learning community, usually referring to the sole in– and output data (which we will refer to as *dataset*), we define the *data space* of a deep learning experiment as follows:

> Definition
> **Data Space of DL Experiments**
>
> We define the **data space** of a deep learning experiment as the entirety of data arising in a deep learning experiment. A **deep learning experiment** denotes the entire model building, diagnosis, and refinement workflow over multiple model iterations to solve a particular task.

This data can be categorized according to its origin, dimensionality, and variability over time. Additionally, we classify the data into the categories *structural*, *scalar*, *high-dimensional*, and *functions*. In the following, we give a detailed characterization of those categories, building the foundation for our proposed framework. Table 4.1 provides an overview.

(D1) Structural ($G = (V, E)$) — Data entities that logically form a graph. The graph structure can be either inherent to the data or originate from the temporal evolution of its entities (cf. a family tree). Following this definition, there are primarily two sources of structural data in the ML workflow, one being the evolution of models over time and the other being the architectural graph of each model.

(D1.1) Tree of Models — Starting from an initial architecture, the iterative diagnosis and refinement process will lead to ever-new model generations with slight modifications compared to the ancestor generation. Addressing **(R1)**, we propose to model this process as a directed graph, with nodes representing model architectures and links indicating a parent-child relationship. It should be noted that this graph does not always stricly form a tree since desirable characteristics of multiple models in the ancestor generation might be merged into a new architecture, embodying multiple inheritance.

(D1.2) Architecture Graph — The architecture graph represents the order of operations applied to data while flowing through the model as a directed graph. This graph representation, targeting **(R3)**, can be of varying granularity; typically, nodes encode the building blocks modern ML frameworks offer for model building, such as layers or functions, while links indicate the data flow between nodes.

Structural data is considered constant over time. While this assumption seems evident for the architectural graph of a model, it might at first be contra-intuitive for the temporal evolution of models. However, the different architectures could have been determined since the beginning of the experiment. They do not depend on training, nor do individual nodes change over time. Therefore, we consider the tree of models final under the most recent model iteration.
(D2) Scalar ($\mathbb{R}$) — Real number describing the model configuration (hyperparameters) or the model state at a single point in time (metrics).

(D2.1) Hyperparameters *Learning rate* or the *batch size* are a form of scalar data describing properties which significantly influence the performance of the model, despite neither being part of the architecture nor the trained instance. They are (usually) only defined once and, therefore, considered constant over time.

(D2.2) Metrics *Accuracy* and *loss* **(R2)** are typical representatives of this category. However, other scalar descriptors might be relevant for debugging, such as a model's execution time or its memory consumption. Metrics are considered variable over time, i.e., changing throughout the training process.

(D3) High-Dimensional ($\mathbb{R}^n$) — Vectors or matrices representing human-interpretable, but also abstract data entities. For example, visual representations of image data can be readily understood by humans, whereas abstract data distributions might require transformations or targeted visualizations to make them approachable by a human. High-dimensional data covers **(R4)**. It often consists of several thousand individual values and therefore places high demands on storage, memory, and calculations. In the

context of the DL workflow, we distinguish different sub-categories of high-dimensional data, depending on their origin:

(D3.1) In- / Output — Data that is fed into the model or matched against the model output. In– and output data describe either single samples or aggregations used for model training and –inference. It is considered constant over time, since the individual samples do not change during training. In– and output data is 1 to n-dimensional. Examples include time series (1D) or images (2/3D).

(D3.2) Weights — Data that is inherent to the current network instance. We refer to weights as all the network variables that are learned during the training process. Therefore, they are considered variable over time. The dimensionality of weights can range from 1D (e.g., biases), over 2D (e.g., dense kernels) to nD (e.g., convolution kernels). Their values are often initialized according to a certain strategy, e.g., by sampling from a Gaussian distribution.

(D3.3) Activations — Data that is dependent both on the network instance and the network input. Specifically, activations are the response of each network entity under the data fed through the network. Since activations combine input data and weights, they change over the training process of the network and, therefore, are considered variable over time. Activations can be human-interpretable by default (e.g., in convolutional networks) or abstract 1 to n-dimensional distributions (e.g., dense networks).

(D4) Function ($\mathbb{R}^n \rightarrow \mathbb{R}^n$) — Mappings between number ranges or matrix shapes. Functions can be seen as the elements forming the *architecture graph*. A function's output is only dependent on its input; thus, functions are considered constant over the training time. In terms of neural networks, we distinguish two sub-categories:

(D4.1) Structural Functions — Functions that modify the shape or structure of data entities with the primary goal to render them compatible with mathematical functions. Examples include concatenation or reshaping.

(D4.2) Mathematical Functions — Mathematical operations, mapping an input to a specific output. Regarding neural networks, many mathematical functions take learned weights as additional parameters. The weights are iteratively adjusted during training to make the function results resemble the training data distribution as close as possible. Examples include matrix multiplications, convolution operations, or activation functions.

While the variability of each of the introduced data categories might at first seem secondary, it is fundamental for the conceptual framework we derive in the following. Particularly, (1) constant data entities can be used as a backbone for variable data and (2) we do not have to resolve the progression over the course of training when visualizing them.

4.3.2 Structuring Design Dimensions for Representation and Interaction

Summary
Sec. 4.3.2

To provide the necessary flexibility for a general-purpose debugging framework, we propose a modular approach for creating *debugging components*. A debugging component is a user-interface element visualizing one of the discussed data entities. Guiding the design of debugging components, we derive different aspects and constraints that have to be considered to make data accessible to the user during the debugging process, called *design dimensions*. Overall, we identify 6 design dimensions that influence the instantiation and appearance of a debugging component.

A *debugging component* is a modular user-interface element visualizing one of the discussed data entities. Following, we describe how a debugging component is instantiated and which dimensions influence its appearance.

Besides the previously covered data space, the systematic debugging of DL experiments involves various other dimensions. Table 4.2 gives an overview of those dimensions, including their characteristics and providing examples for each. Depending on the use case and the individual machine learning workflow, other important characteristics may arise, complementing the pre-identified aspects. In the following, a short description of each dimension is given to substantiate the dimensions and their characteristics.

4.3.2.1 Design Dimensions for Data Representation and Interaction

(I1) Task — The goal of the analysis and, thereby, the debugging process. The analysis task is strongly related to the underlying machine learning task, i.e., the task for which the model is designed. For example, in a setting where high classification accuracy is demanded, **model assessment** might be the primary analysis task. In contrast, if the model is designed to be used in a security-critical environment, formal **verification** will likely be of absolute priority. The analysis task is also influenced by the stage of the machine learning workflow. Usually, multiple architectures are compared against each other during the initial model selection, focusing on model **assessment** and **–comparison**. In a later stage, when the rough architecture is finished and should be fine-tuned on the machine learning task, **correctness checking** is needed to verify if the model behaves according to human intention. For example, local XAI methods might be used in this stage, or distributions in the latent space of the model could be investigated.

(I2) Level of Abstraction — The granularity in which the model is inspected. The level of abstraction strongly depends on the stage of the debugging process. In the beginning, an overview over **multiple models** might be required, e.g., to assess and compare their performance. Then, exciting details like abnormalities in their architectures or

Dimension	Characteristics	Examples
Data	*See section 4.3.1, "Capturing the Data Space of Deep Learning Experiments".*	
Task (I1)	Assessment	Assess model quality
	Verification / correctness check	Verify that model behavior follows human intentions
	Comparison	Compare architctures
Level of abstraction (I2)	Multi-model	Model change tracking
	Single model	Quality measures
	Layers / Units	Distributions
	Weights / Neurons	Activation maximization, deep dream
Processing (I3)	None (raw)	Images, text, scalars
	Transformation	Re-shaping, projection
	Aggregation	Binning, clustering
	Statistical descriptors	Variance, min, max, density estimation
Representation (I4)	Visualization	Charts, histograms, images
	Verbalization	Text, tables
Dependencies (I5)	Dataset	Image, text, categorical
	Model	Feed-forward, recurrent
	Layer	Dense, covn., operation

Table 4.2: The design dimensions influencing appropriate data representations. Despite them being mostly orthogonal to each other, dependencies might arise upon their combination. For example, while **binned** values could be represented through **visualization** or **verbalization** and could be used for **assessment** or **comparison**, they can never be applied to **structural** data.

a high loss can be tracked down into **single models**. For an even closer debugging, architectural details or kernel distributions of single **layers** might be observed to understand where a particular model behavior originates. Finally, for highly specialized use cases, even an inspection of single **neurons** or **weights** might be needed, e.g., to assess the performance of pruning strategies or by using advanced XAI techniques, such as activation maximization with a deep dream approach.

(I3) Processing — Transformations to bring data into a human-readable format and reduce storage– and compuational complexity. For example, a weight matrix constitutes an abstract data distribution, being opaque to a human in its **raw** form and, therefore, requiring appropriate abstraction. Usually, relevant features should be preserved and emphasized while reducing the total amount of data, which strongly relates to the analysis task: if the analyst is only interested in the range of values in the weight matrix, minimum and maximum as **statistical descriptors** already fulfill the analysis goal. Ideally, the appropriate processing transforms the data into a format that directly answers the analysis question (e.g., min or max to assess the value range), is inherently

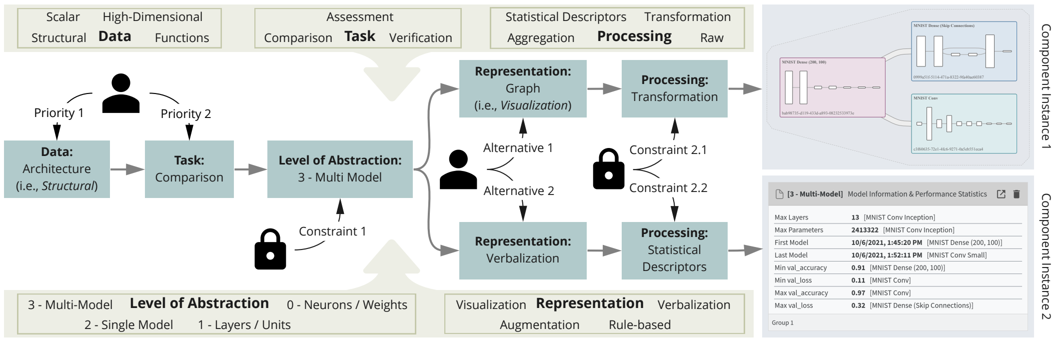


Figure 4.2: An exemplary instantiation of a debugging component. The user can select characteristics for preferred dimensions, which might lead to constraints in other dimensions. Eventually, all dimensions are determined by iterating this process, and the component can be created.

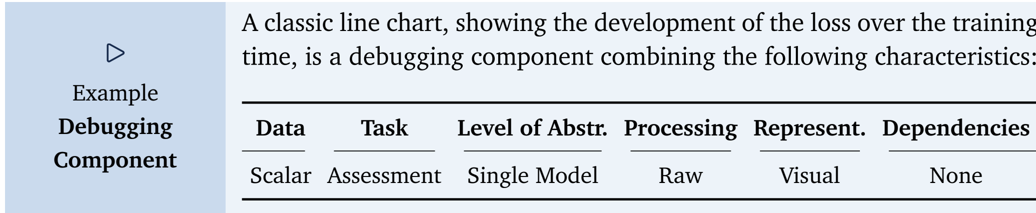
interpretable by a human (e.g., images or text), or conforms with the human's mental model of the data. For example, the distribution of latent activations could be **aggregated** using binning, telling the human if the data distribution follows his intuition on how the values should ideally be distributed in the latent space.

(I4) Representation — How the data is presented to the user. The ideal representation depends on various factors, such as the analysis task or the kind of data. Furthermore, human perception plays an essential role: data types that are inherently interpretable by a human, such as graphs or images, benefit from a **visual** representation, while text or tabular data are ideally represented as **verbalization**.

(I5) Dependencies — Some debugging techniques come with limitations regarding their applicability, referred to as dependencies. The characteristics of the other dimensions determine the dependencies; therefore, they form an inferred dimension, denying direct user control. For example, a **dataset dependency** arises upon the inspection of activations, which only exists under input data. This contrasts with an inspection of, e.g., the model architecture, which already exists at design time and, therefore, is **dataset agnostic**. Other inspections might only apply to a specific type of model or layer, forming **model–** or **layer** dependencies, respectively. Examples are investigating time steps in RNNs or the image-based inspection of convolution kernels.

4.3.2.2 Combining Design Dimensions

To combine the identified dimensions into an overarching guideline for the systematic debugging of machine learning experiments, we propose a systematization based on *debugging components*. A debugging component is a modular user-interface element, giving insights into one of the discussed data categories. To instantiate a debugging component, one or multiple identified dimensions must be set to a particular characteristic, determining its final appearance.



Example **Debugging Component**

A classic line chart, showing the development of the loss over the training time, is a debugging component combining the following characteristics:

Data	Task	Level of Abstr.	Processing	Represent.	Dependencies
Scalar	Assessment	Single Model	Raw	Visual	None

Our proposed framework describes the creation of debugging components as a modular approach, comparable to choosing elements from a toolbox and combining them. Typically, the user has clear preferences on which aspects the debugging process should focus. By choosing characteristics for preferred dimensions, the user can narrow down the appearance of the debugging component. This process must be iterated until all dimensions are fixed by the user or by constraints arising from combinations of already-defined dimensions. Therefore, with increasing specificity, more and more options will either be fixed or get “greyed out”, until all dimensions are set to a particular value, determining the final appearance of the debugging component.

Figure 4.2 shows an exemplary instantiation of a debugging component, following the described process. Initially, no constraints are present, and the user can freely determine his preferred dimensions. The **data** dimension is set to **structural** in the example since the user wants to assess the model architecture. Next, **comparison** is set as the task, which infers the first constraint: for a comparison, multiple models must be considered simultaneously. Therefore, a degree of freedom is removed in the **level of abstraction** dimension, automatically inferring the **multi-model** level. There are still degrees of freedom in the dimensions **representation** and **processing**. The user can now decide between a **visual** and a **verbal** data representation, which in both cases automatically constraints the **processing** dimension to **transformation** or **statistical descriptors**, respectively. This leads to all dimensions being fixed, fully determining the debugging component’s appearance and allowing its instantiation.

4.3.3 Navigating the Data Space through Global Mechanisms

Summary **Sec. 4.3.3**

Complementing the design dimensions that determine the appearance of a debugging component, we propose a set of global mechanisms to navigate the data space. The mechanisms are suggested functionalities a system for debugging DL experiments should provide. These mechanisms set our framework apart from existing debugging systems by proposing a comprehensive way to locate entities of interest and access their underlying data through them.

▷ Example **Global Mechanisms vs. Existing Systems**	Existing systems, such as TensorBoard, treat the logged data as independent entities that are not connected to each other. E.g., one tab shows the architecture graph, another tab shows scalars, and a third tab shows images. In contrast, our framework leverages the natural connection between the entities. E.g., models are composed of layers, which are composed of kernels and operations. Training metrics are related to a model, activations are related to layers, and weights are related to kernels. Therefore, our framework uses the architecture graph to navigate to the entity of interest, e.g., a convolutional layer. From there, the user can access the layer's weights and activations by applying tools to them, revealing their underlying data.

In the following, we identify global mechanisms which the analysis toolchain can implement to make the previously structured data space explorable by the model developer. Although these mechanisms should are considered independent of each other, they become particularly effective in combination. Figure 4.3 shows an overview of the mechanisms and how they work together to navigate the space.

(M1) Units of Analysis *(UoA)* — The data space has to be recursively segmented into logically complete subcomponents, making them accessible by the analyst. For example, depending on the analysis task, the analyst might be interested in a single model, a specific layer, or even a single neuron of a model, representing units of analysis of different granularity. Analogously, such units of analysis can be determined for the majority of the identified data categories and sub-categories.

(M2) Structural Backbone — Linking abstract data to real-world entities facilitates integration into the analyst's mental model. For this, structural data can be used as a backbone, providing access to units of analysis in a navigable graph representation of the abstract data entities. Besides the visual representation of the model architecture, which is prevalent in existing tools [105] and related works [35, 208, 104] and tracking of models in version control systems [205], our framework proposes to additionally visualize the evolution of the models themselves in a tree-like structure, offering a powerful tool for model tracking and change analysis.

▷ Example **Applying Tools to UoAs**	For example, an analyst might want to debug a vision model by applying LRP [23] to its layers. He navigates to the model and layer of interest *(∼ unit of analysis)* by descending into a visual representation of the model's architecture *(∼ structural backbone)*. From there, he can apply the LRP tool to the layer, letting him select an input sample and automatically visualize the resulting saliency map.

(M3) Levels of Abstraction — The elements of the structural backbone inherently form a hierarchy, i.e., some units are sub-components of other units. This hierarchy

is implemented in our framework as different levels of abstraction. The analyst can go from overview to details by navigating over these levels while keeping the global context. For example, if the analyst is interested in a specific filter kernel of a convolution operation, the graph can be navigated over namespaces, layers, and operations down to the kernel of interest.

(M4) Search (Highlighting) — The vast data space spanned by machine learning experiments renders search mechanisms imperative. By searching for certain UoA types and, subsequently, highlighting them in the structural backbone, entities of interest can be effortlessly spotted and tracked down. For this, the highlights have to propagate through the structural backbone up to the highest levels of abstraction. E.g., by searching for convolutional and dense operations and propagating the results up to the tree of models, the analyst can effortlessly distinguish between dense and convolutional model architectures as an entry point to the analysis.

(M5) Interestingness — Complementing the search mechanism, which helps to locate data whose shape is known beforehand, interestingness detection can point the user to irregularities in the data [209], covering **(R5)**. For example, the analyst might be interested in kernels whose distribution diverges significantly from a baseline. Interestingness measures could identify such entities as abnormal and point the user to the corresponding unit of analysis. Automated interestingness detection is essential since manual annotation is not feasible due to the large amount of data to cover.

Analogous to the accepted classifications of interestingness measures in the context of data mining [209, 210], we provide the following definition in the context of debugging neural networks.

Definition **Interestingness in DL Debugging**	In the context of debugging neural networks, we define **interestingness** as a quantifiable measure of **abnormality in the data**, described as the **deviation from a specified baseline**. Depending on the type of entity whose interestingness is measured, the baseline can be a fixed value, a distribution, or a rule set. The baseline can either be inferred from the data domain (objective), learned from other data or a ground truth (subjective), or defined by the users (user-defined).

In the following, we provide examples of interestingness measures in the context of debugging neural networks to concretize the provided definition.

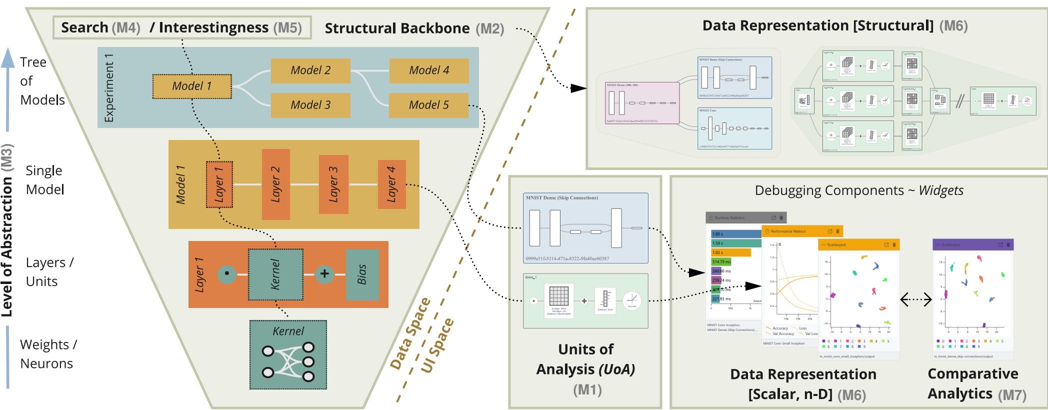
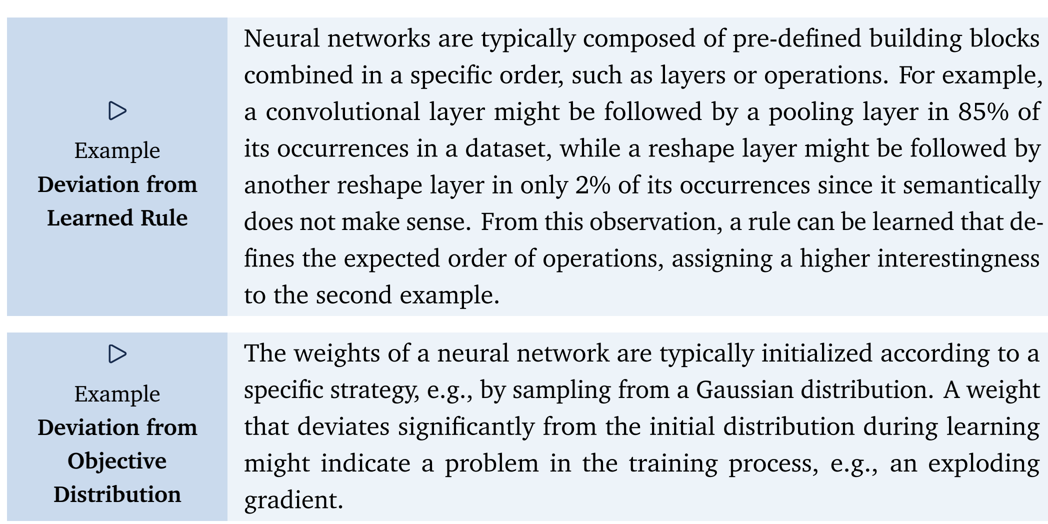


Figure 4.3: Overview of the global mechanisms to navigate the data space. The structural data provides a backbone for navigation over multiple levels of abstraction. Entities of the structural data are referred to as units of analysis, which can have multiple debugging components with different characteristics attached. Interestingness measures and filters guide through the structural backbone to relevant units of analysis.



▷ Example **Deviation from Learned Rule**	Neural networks are typically composed of pre-defined building blocks combined in a specific order, such as layers or operations. For example, a convolutional layer might be followed by a pooling layer in 85% of its occurrences in a dataset, while a reshape layer might be followed by another reshape layer in only 2% of its occurrences since it semantically does not make sense. From this observation, a rule can be learned that defines the expected order of operations, assigning a higher interestingness to the second example.
▷ Example **Deviation from Objective Distribution**	The weights of a neural network are typically initialized according to a specific strategy, e.g., by sampling from a Gaussian distribution. A weight that deviates significantly from the initial distribution during learning might indicate a problem in the training process, e.g., an exploding gradient.

(M6) Appropriate Data Representation — Only a small share of the data arising in machine learning experiments is inherently interpretable by humans. Often, the data encodes complex relationships (e.g., graphs), is of high dimensionality (e.g., kernels), or of abstract type (e.g., time series). Therefore, processing and representation have a great impact on the accessibility of the data. Ideally, the data representation helps the user focus on the relevant details while insignificant or redundant information is discarded. Furthermore, the representation should be targeted towards human perception. For example, humans have an intuitive understanding of images and text; in contrast, making sense of a series of sensor values is inherently complicated.

(M7) Comparative Analytics — Comparing data across different units of analysis is a fundamental use case in the debugging workflow, allowing the analyst to evaluate them against each other or reason over abnormality. Therefore, it is essential to enable

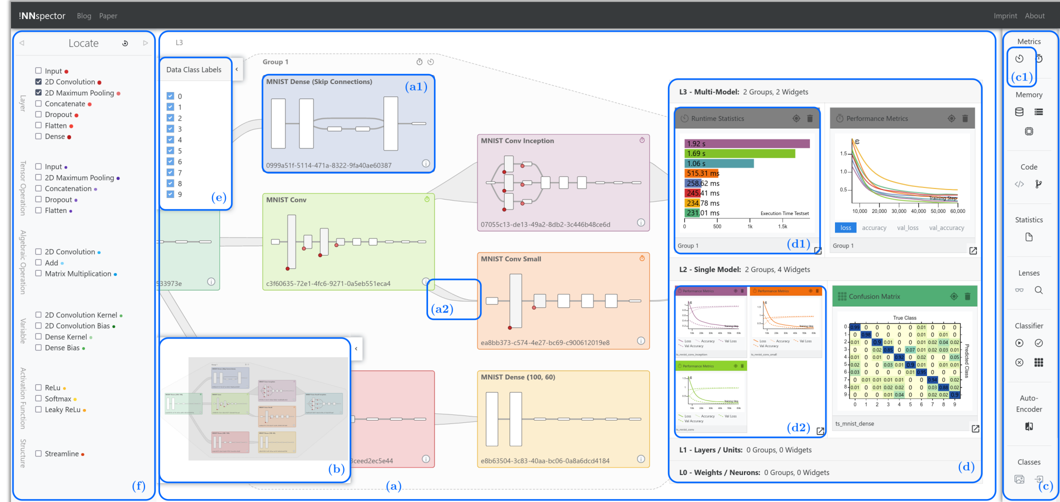
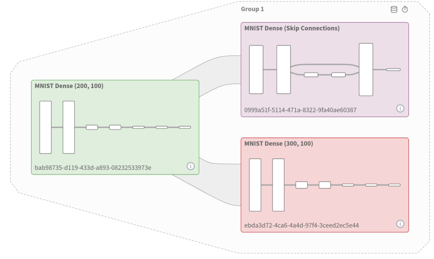


Figure 4.4: The iNNspector frontend. It is built around the inspection panel (a), showing nodes (a1) and links (a2) of the structural backbone on the current level of abstraction. The minimap (b) helps to navigate the viewport. Tools (c1) from the Toolbox (c) can be applied to units of analysis in the structural backbone to create widgets (d1), showing underlying data. Widgets are arranged in the widget panel (d), where they are organized according to their level of abstraction. Semantically related widgets can be combined into groups (d2). Widgets showing class-dependant data can be constrained to certain classes using the global class selector (e). The localization and interestingess panel (f) provides tools to identify units of interest.

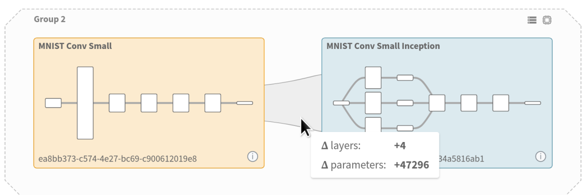
the concurrent inspection of multiple UoAs, e.g., by implementing side-by-side views. In this scenario, multiple data representations from remote units of analysis might be simultaneously visible, e.g., when comparing two layers originating in different models. Therefore, navigational patterns for retrieving the underlying unit of analysis have to be provided, allowing the user to jump to a debugging component's respective unit of analysis.

4.4 The iNNspector System for Systematic Model Debugging

Materializing the before-proposed conceptual framework, we present the iNNspector system for systematic model debugging. More than two years of conceptual ideas and implementation work have gone into the system. The system is driven by three primary components: the custom keras checkpoint, collecting data for iNNspector during training, the backend, providing the data of the experiments via REST API, and the frontend, providing the actual iNNspector user interface.



(a) First experiment, with 3 models.



(b) Second experiment, with 2 models.

Figure 4.5: Inspection panel on L3, 'Multi-model'. Experiment (a) has three models and experiment (b) two, each model represented by a unique color. Edges denote parent-child relationships, with tooltips and edge width indicating changes between model pairs.

4.4.1 Application Walkthrough

In the following, we give a description of the iNNspector interface and its components in the form of an elaborate application walkthrough. Thereby, we illustrate how iNNspector meets its claim of being a comprehensive system facilitating the systematic debugging of DL models. By consequently linking system features to the respective data category (see section 4.3.1), design dimension (see section 4.3.2), and global mechanism (see section 4.3.3), we show how the system design is anchored in the requirements analysis (see section 4.2) and the foundations compiled into our conceptual framework (see section 4.3).

Inspection Panel — iNNspector is built around a central graph view, referred to as *inspection panel* (figure 4.4a). It visually represents the structural data of the experiment, i.e., the tree of models, the model's architectures, and subsets of the weight-neuron network. Thus, it implements the structural backbone **(M2)** and builds the primary component to locate units of interest **(M1)** over different levels of abstraction **(M3)**. Starting with the multimodel view (4.4.1.0.2, L3), showing the tree of models as they evolved during the experiment, the graph can be navigated to lower levels: by double-clicking a model and, subsequently, a layer, the user can descend over the single-model-view (4.4.1.0.3, L2) down to the neuron-weight-network view (4.4.1.0.4, L1). The following paragraphs elucidate the specifics and functionalities of each layer.

Multi-Model View [L3] — The highest level of abstraction is the *multimodel view*. It shows all models in the iNNspector system, grouped into *experiments*. An experiment denotes a connected subset of models, i.e., a subset where each model shares a parent-child relationship with at least one other model. Figure 4.5 shows two experiments, the first consisting of three, the second of two models. Each experiment is surrounded by a dashed outline, which forms a unit of analysis itself. Each model is assigned a color based on a sequential color scale, which is re-used by other parts of the system to mark the part as belonging to a certain model, e.g., when descending into lower levels of

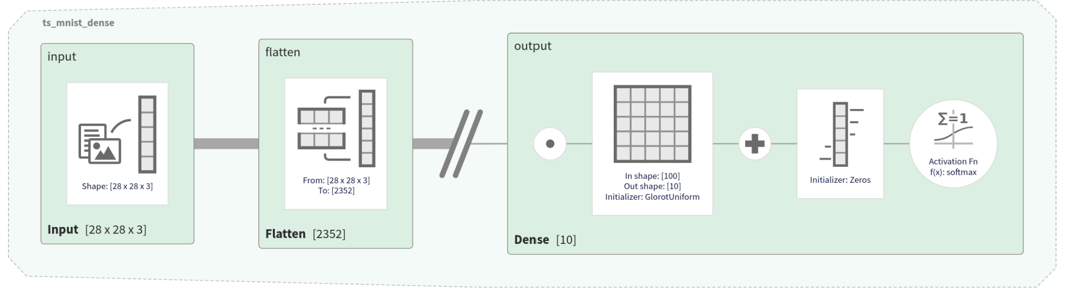
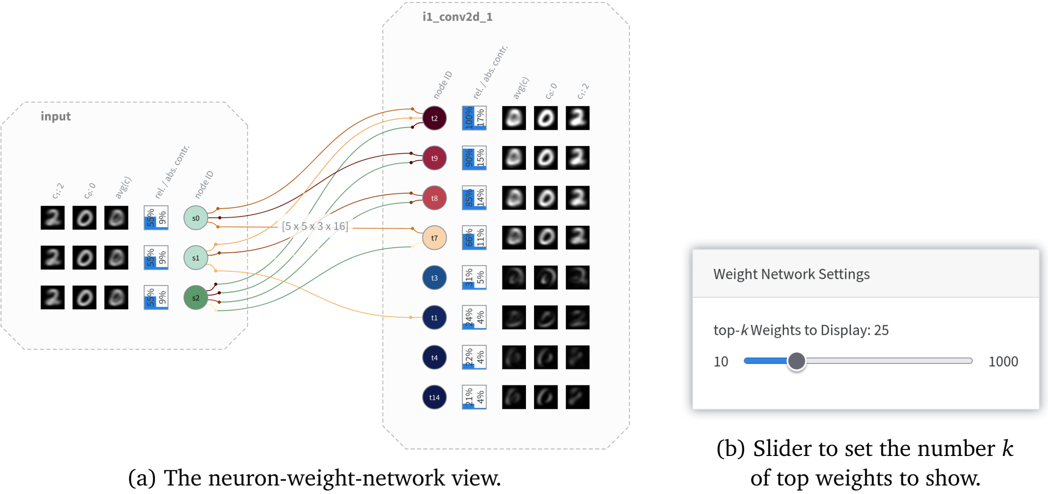


Figure 4.6: Inspection panel on L2, “Single Model”. The view shows the architecture of a model, with each box representing a layer and the edges denoting the data flow between layers. Elements inside the layer boxes represent operations applied to the layer input.

abstraction. Each model is depicted by a box, containing its friendly name, which is set by the programmer at development time (see 4.4.4.1.1, “iNNspector Keras Logs”), its unique ID, and an abstract representation of its architecture. Each tree of models is arranged from left to right, with the leftmost model being the initial architecture and descendant models being ranked according to their depth in the tree. The edges indicate a parent-child relationship, which is defined during development time, either manually by the programmer or by automatically branching models using the available tool from the toolbox. The change in the edge width between models denotes the relative change in the number of trainable parameters, with the absolute values being shown in a tooltip when hovering the edge.

Single Model View [L2] — By double-clicking a model, the user can descend by one level into the *single model view.* Figure 4.6 shows the (shortened) single model view of the upper left model visible in figure 4.5. It provides a graph visualization of the model’s architecture, with each box representing a layer of the model and the edges denoting the data flow between layers. The dashed outline denotes the model as a whole. All layers universally show their name in the top left, and their type and output size in the bottom left. Additionally, layers contain one or multiple inner elements, encoding information specific to their type and configuration. These inner elements form a graph on themselves, which represents the way mathematical functions and variables are subsequently applied to the layer input. E.g., the dense layer in figure 4.6 starts with a matrix multiplication ⊙ of the input with a kernel ▦, continues with adding ✚ a bias, and finally applies a softmax activation to the result. The pictograms give a visual impression of the respective operation or variable, rendering the system useful for less experienced model developers or for educational purposes. Additionally, operations and variables are augmented with information relevant for model development, such as input and output shapes, kernel initializers, or filter shapes.

Neuron-Weight-Network View [L1] — By double-clicking a layer on L2, the user can descend to the final level L1, the *neuron-weight network view*, of which an example is



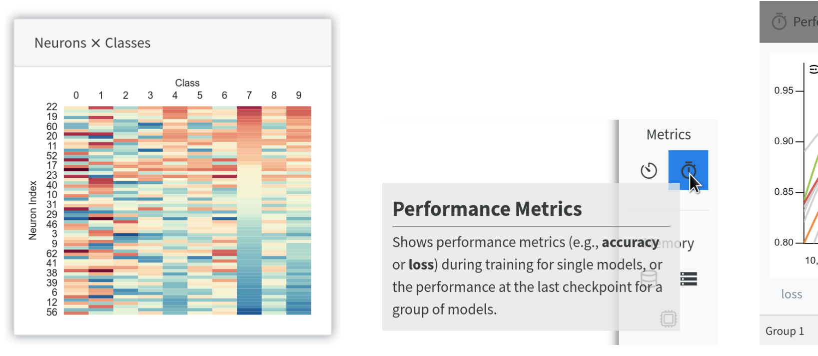
(a) The neuron-weight-network view.

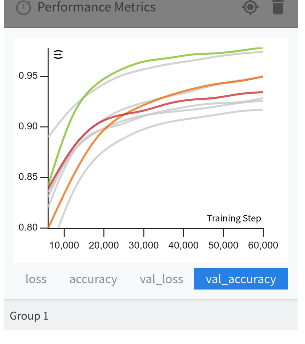
(b) Slider to set the number k of top weights to show.

Figure 4.7: Inspection panel on L1, "Neuron-Weight-Network". The view shows the top-k most significant subset of neurons for both layers, according to their mean activation under the current class selection. The edges between the neurons represent the elements of the weight matrix connecting two neurons through a matrix multiplication or a convolution operation.

shown in figure 4.7a. Like the name suggests, it focusses on the weights connecting the previous to the current layer. For this, we visualize a subset of neurons for both layers, sorted by their mean activation under the current class selection (see 4.4.1.0.8, "Application Walkthrough"). The neurons are colored according to a linear color scale, mapping the lowest activation to dark blue and the highest activation to dark red. Edges between the neurons represent the largest k weights, i.e., the element of the weight matrix which connects the two neurons through a matrix multiplication or a convolution operation. The value for k can be set using a slider, which is shown in figure 4.7b. Since the number of neurons in a layer can quickly grow to the order of several thousands, we filter for the ones that (1) are connected through at least one of the top-k weights, or (2) feature either one of the 10 highest or 10 lowest mean activations. Neurons and edges can be highlighted by hovering or clicking them, which makes it easier to observe single connections in large networks. Besides the activation color, we augment the neurons with additional information, such as their relative and absolute share of the summed activations currently viewed. Furthermore, for activations having an image-interpretable format (i.e., two-dimensional grayscale or three-dimensional coloscale), we show their mean activation c_i for $i = 0, \dots, n-1$ for the n currently selected classes, as well as an average over the selected classes $\text{avg}(c) = \frac{1}{n} \sum_{i=0}^{n-1} c_i$.

Since the neuron-weight-network can only visualize a subset of neurons and weights and always averages over the current class selection, we augment the L1 view with the *Neurons × Classes panel*, shown in figure 4.8a. It provides a matrix view showing the activation of each individual neuron with respect to each class of the datset, rendering it especially useful for the debugging of failed classifications and disentanglement of latent





(a) Neurons × Classes panel. (b) The iNNspector toolbox. (c) An exemplary widget.

Figure 4.8: Panels, tools, and widgets in the iNNspector UI. (a) A L1-specific panel showing the activation of each neuron with respect to each class. (b) The toolbox, providing tools to investigate units of interest. The tools can be applied to compatible elements in the inspection panel to create widgets. (c) An exemplary widget, showing the activation of a neuron with respect to each class.

spaces. By clicking the header of a class column, all columns are re-sorted according to the clicked class, enabling correlation analysis across classes. Notably, with decreasing level of abstraction, the data space we have to cover shrinks significantly. Therefore, more data can be shown by specialized panels (e.g., *Classes* × *Neurons*) and the structural view itself, reducing the number of tools that are needed to show underlying data.

Toolbox, Widgets — The **toolbox** (figure 4.4c) provides a variety of tools to investigate and annotate units of interest, extend the tree of models by new model variations, and – most importantly – visualize the underlying data of units of analysis. Each tool is represented through a small icon, which, when hovered, offers a detailed description of the tool's functionality, as displayed in figure 4.8b. After selecting a tool, it stays active until applied to a unit of analysis, e.g., a model or a layer. The tools in the toolbox resolve several dependencies **(I5)** by narrowing their applicability to specific UoA types and levels of abstraction. E.g., the *Histogram* tool, which creates a histogram visualization of a data distribution, can only be applied to UoAs comprising high-dimensional data **(D3)**, such as kernels or activations of a layer. In most cases, applying a tool to a UoA results in the creation of a **widget** (figure 4.8c). A widget offers visual or verbal representations **(I4)** of the underlying data of a unit of analysis. This data can be scalar **(D2)** or high-dimensional **(D3)**, rendering appropriate processing **(I3)** essential. Therefore, each tool defines which data to query and how to transform this data before forwarding it to the newly created widget. The transformation pipeline is expressed as a list of transform operations that are consecutively applied by the backend to the queried data before returning them. The data querying mechanisms follow a standardized format, making the system effortlessly expandable by custom tool. For technical details on the data querying routes and the transformation grammar, see section 4.4.4.2.

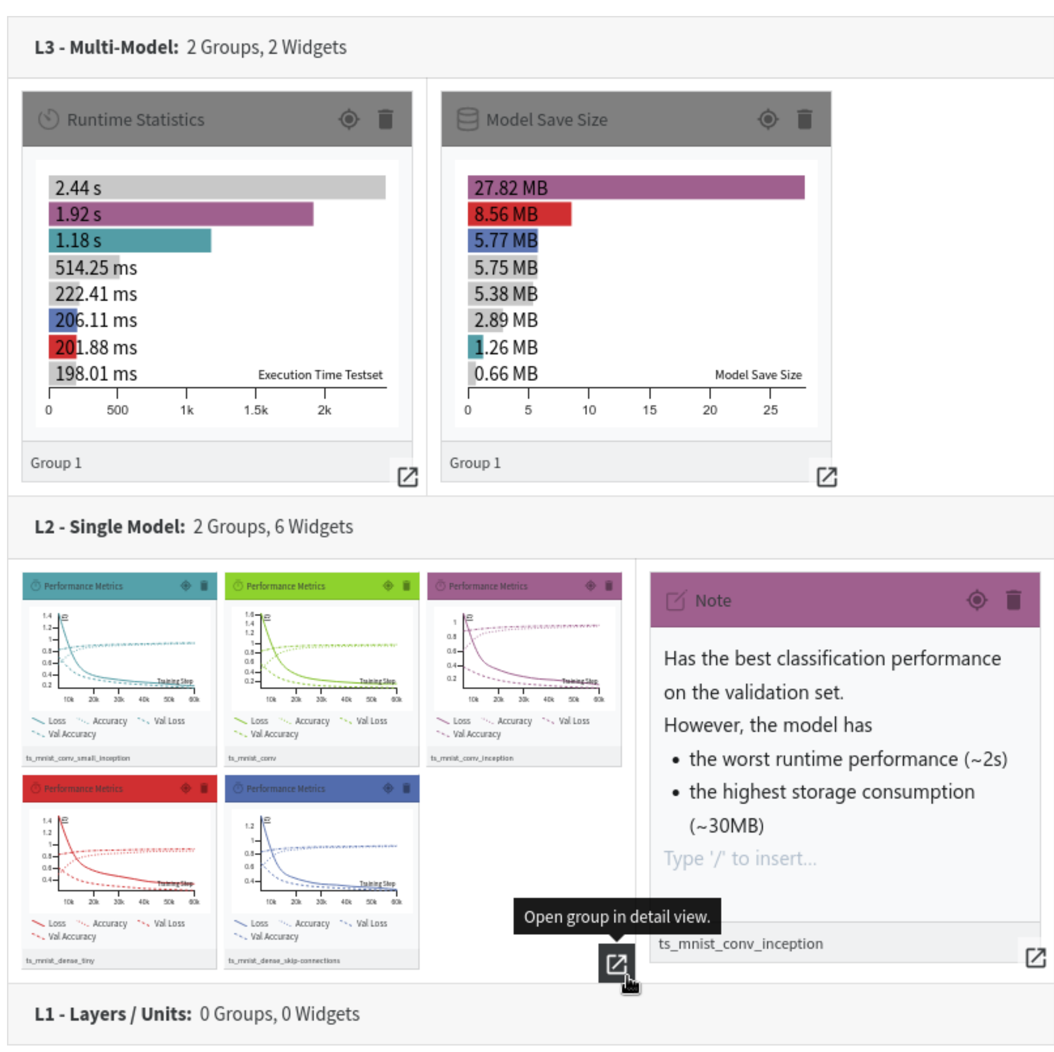
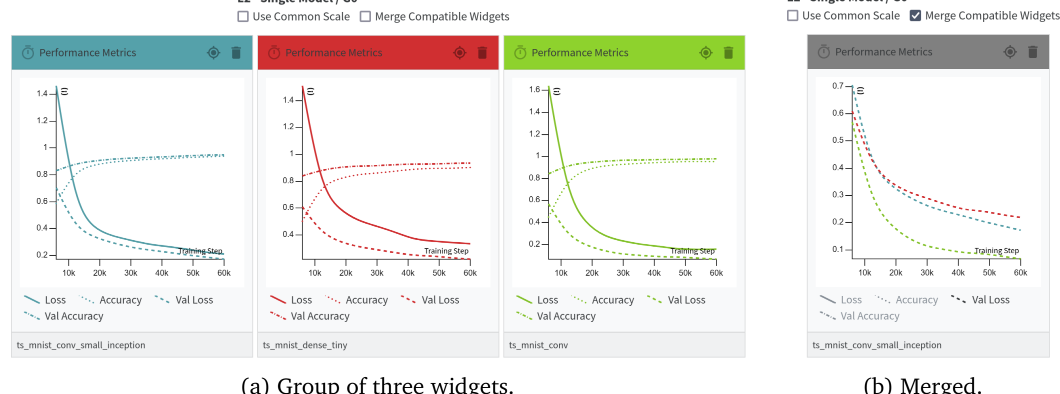


Figure 4.9: The iNNspector widget panel. Widgets are organized according to their level of abstraction. They can be grouped by dragging and dropping them onto other widgets or groups.



(a) Group of three widgets.

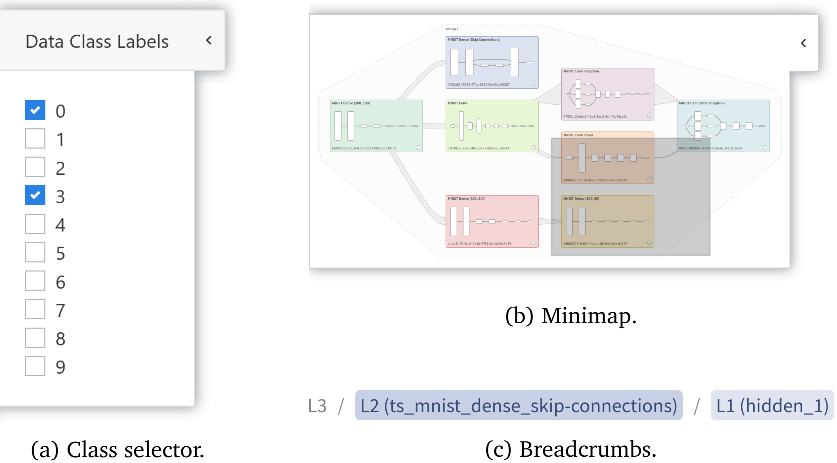
(b) Merged.

Figure 4.10: iNNspector's widget group view. (a) Grouped widgets can be opened in a detail view, showing them side-by-side in an enlarged page overlay. (b) Widgets featuring compatible data can be chosen to either use a common scale or be merged into a single meta-widget.

We provide a comprehensive set of pre-defined widgets, supporting a variety of use cases and analysis scenarios. The widgets can be classified into three groups: (1) generalizing representations, (2) data-specific representations, and (3) functional. **Generalizing representations** provide typical charts for common types of data. While they set strict conditions on the format of the data, they do not introduce any further dependencies **(I5)**. For example, we provide linecharts for time-dependent, scalar data or bar charts for single-time scalar data. Complementary, we offer different types of histograms for high-dimensional data. **Data-specific representations** are charts specialized to a certain type of model, layer, or dataset. For example, for classifiers, we include a visualization of the class probability distribution, while for image-to-image models, we visualize the difference between input and reconstruction. Both only work for the stated type of model. Finally, functional widgets do not visualize underlying data of the experiment, but provide information for model management and documentation of the analysis process. For instance, the *Note* tool creates a widget featuring a markdown editor, while the *Branch Model* tool generates a new iNNspector header, denoting a new parent-child relationship in the tree of models. For details on the iNNspector header, refer to paragraph 4.4.4.1.1, "iNNspector Keras Logs".

Widgets can be extended through arbitrary interactions, e.g., to filter or exclude data series, or constrain the widget to a subset of UoAs. Particularly, this includes linking and brushing of widgets and their contained data across all components of the iNNspector system. See "Application Walkthrough" for more details.

Widget Panel — Widgets are organized in the **widget panel**, depicted in figure 4.9. The panel is implemented as a vertical accordion UI element, with each accordion flap representing one level of abstraction. Upon their creation, widgets are inserted into the flap of their corresponding unit of analysis lies on, reflecting the structural hierarchy



(a) Class selector.

(b) Minimap.

(c) Breadcrumbs.

Figure 4.11: Configuration and navigation panels in iNNspector. (a) The class selector panel lists all classes occuring in the dataset, allowing to toggle them. The class selection affects the data of all widgets in the widget panel that are class-dependent. (b) The minimap indicates the viewport of the iNNspection panel, guiding the user in large experiments or models. (c) Breadcrumbs show the path to the UoA currently focussed in the iNNspection panel over the different levels of abstraction.

also in the widget panel and, hence, helping the user to locate searched-for widgets. This effect could be proven in our evaluation study (see section 4.5.2.2). To prevent the cumulative cluttering of the widget panel during the advance of the analysis process, widgets can be **grouped** by dragging and dropping them onto other widgets or groups. Besides visually sorting semantically related widgets, groups also can be opened in a detail view, showing them side-by-side in an enlarged page overlay. In the group view, depicted in figure 4.10a, widgets featuring compatible data can be chosen to either use a **common scale** or be **merged** into a single meta-widget. Scaling and merging makes the data better comparable across units of analysis. Figure 4.10b shows the same group as figure 4.10a but with the widgets merged into a single meta-widget.

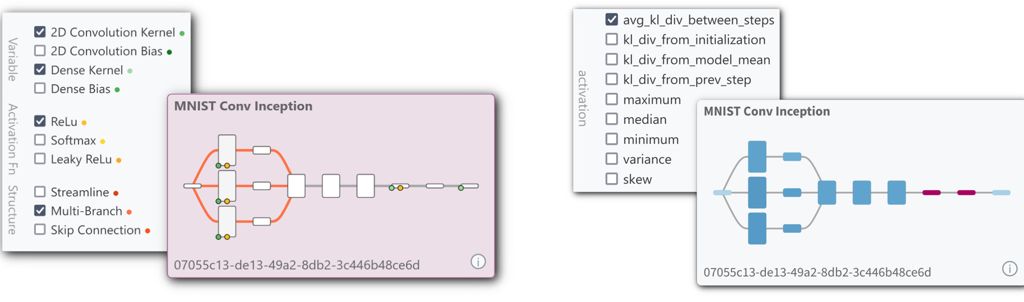
Linking and Brushing —To emphasize connections between related entities, we consistently implement linking and brushing over all parts of the system. On the component level, this affects links between UoAs, widgets, and filters. E.g., hovering a layer in a model's architecture results in a visual highlight of all widgets relating to the layer, and vice versa. On the data level, different representations of the same data-point or dataset sample are linked across the system. E.g., when hovering the projected activation for a certain dataset sample in a scatterplot widget, the datapoints relating to the same dataset sample are highlighted in all other widgets, enabling an interactive comparison across layers or models.

Class Selector Panel — The *class selector panel* lists all classes occuring in the dataset, allowing to toggle arbitrary classes. After modifying the class selection, all components of the system are updated to adhere to the selected classes. Particularly, this affects

(1) the appearance of the iNNspection panel on L1, constraining the structural and high-dimensional data displayed to the selected classes, and (2) the data of the widgets in the widget panel. The only exceptions are widgets showing data where the datapoints are not directly associated to a class (like they are, e.g., for activations), but which would still change under different classes (compared to, e.g., weights, which are class independent). For such widgets and n dataset classes, we would have to pre-compute and store $2^n - 1$ possible combinations. An example for such a widget is created by the "Performance-Metrics"-tool: metrics like loss or accuracy change under different dataset classes, however, since those values are computed at training time, we would either have to pre-compute the data for arbitrary combinations, resulting in huge runtime and storage overheads, or re-evaluate the model just-in-time, affording access to the full training and testing dataset, as well as information on the computed metrics. We mark those widgets with a ⚠-symbol in the widget header.

Navigation Functionalities — For efficient exploration of the structural data space, we include additional functionalities that help the user to keep an overview and locate themselves in the levels and graphs. A minimap, depicted in figure 4.11b, indicates the viewport of the iNNspection panel, i.e., the section of the panel that is currently visible in the browser window. By clicking a point on the minimap, the vieport automatically centers at that position, allowing the user to quickly jump to an area of interest, which is particularly useful for large model architectures on L2. Besides the minimap providing orientation in the x-y-plane, breadcrumbs show the path to the UoA currently focussed in the iNNspection panel over the different levels of abstraction. Figure 4.11c shows the breadcrumbs while having the layer "hidden_1" of model "ts_mnist_dense_skip-connections" focused on L1. Clicking an element in the breadcrumb path directly teleports the user to the respective level of abstraction. A similar functionality is offered by the ◈-button in the header of all widgets. By clicking it, the system automatically jumps to the level of abstraction and the UoA the widget belongs to, regardless of the current position.

Localization — The structural backbone **(M2)** might grow to significant size and depth for large experiments and complex model architectures. Therefore, iNNspector includes a panel allowing to search **(M4)** for and locate certain UoA types and architectural structures. Figure 4.12a shows a segment of the localization panel, listing different UoA types occuring in the currently visible models. By toggling them, small badges in the inspection panel indicate the location of respective UoAs in the structural data. For UoAs hidden in lower levels of abstraction, badges are propagated upwards to the current level, helping the user to track down UoAs of interest. Besides UoA types, the user can also search for common structures in the model architecture [107], such as multi-branches (c.f., Inception, Szegedy et al. [211]) or skip-connections (c.f., ResNet, He et al. [212]).



(a) UoA types and –badges.

(b) Interestingness types and –annotations.

Figure 4.12: Localization and interestingness functionalities in iNNspector. (a) The localization panel lists different UoA types occuring in the currently visible models. By toggling them, small badges in the inspection panel indicate the location of respective UoAs in the structural data. (b) The interestingness panel colors UoAs according to their interestingness scores.

Interestingness — Complementing the filters, which work exclusively on a structural level, we implement automated **interestingness measures (M5)** on a data distribution level. Particularly, we compute a variety of statistical descriptors over the high-dimensional data **(D3)**, including *skew*, *variance*, *minimum*, and *maximum*, as well as the divergence of the distribution shape from different baselines. The interestingness panel in figure 4.12b allows to toggle the different descriptors, which are then visualized in the inspection panel. Interestingness is computed per UoA on the lowest-level where data occurs, and then propagated upwards over the structural backbone. Since the statistical descriptors are only comparable within one variable type (i.e., activation, dense kernel, conv2d bias), we compute the values for each variable type separately. The values are normalized and aggregated over layers and models. Visually, we annotate the interestingness by re-coloring the UoAs on a linear color scale, ranging from blue for low to red for high interestingness values. Interestingness can directly point the analyst to annomalies in the data without the tedious, manual inspection of large parts of the data space, providing an entry-point to the distribution analyis.

4.4.2 Design Rationales

Throughout our two-plus-year development process, we went through various design iterations until the system reached its current form. In the following, we will elaborate on the more prominent design rationales that shaped the iNNspector frontend.

Widgets — As proposed by our framework and, consequently, implemented by iNNspector, we use the structural backbone to localize UoAs and visualize their underlying data. Following this principle of connecting abstract data to semantically related, tangible visual representatives, we attached the widgets directly to their respective UoAs in the first version of the system. While this emphasized the affiliation between widget and

UoA, we found several issues with the design. For example, the interface got increasingly crowded during the inspection, and there was no natural way to create groups of widgets. Furthermore, the close link between widget and UoA was a disadvantage for comparative tasks over multiple levels of abstraction: where would we put a widget if its corresponding UoA is not currently visible in the inspection panel? Therefore, we decided to organize the widgets in a global panel and visually link them through labels and colors. In our evaluation study (see section 4.5.2), our users reported liking how widgets are organized in the widget panel.

Global Class Selector — The class selector is implemented as a global panel instead of a per-widget class selection. In our different design iterations, we experimented with different mechanisms to constrain visualizations to specific classes, advancing from having no class selection over per-widget, single-class selection to the current, global multi-class selector. It best supported the debugging of classification use cases and prevented confusion with a per-widget class selection when comparing across different widgets.

Toolbox — The toolbox is a typical pattern in existing systems (e.g., image editing– or CAD software). It provides a straightforward and tidy interface to complex functionalities, is easy to learn and use, and can adapt to changing contexts by only showing the currently applicable tools.

Levels of Abstraction — In the first iteration of the system, multiple inspection panels could be spawned simultaneously, each of which could be on a different level of abstraction. Through this, we tried to resolve the previously discussed problem of simultaneously inspecting widgets in different models and levels of abstraction, back when widgets were still directly attached to their respective UoAs. However, this led to a loss of global context and crowded the screen with different representations of the same structural data. Therefore, to keep focused and preserve the global context, we moved to strictly hierarchical navigation over levels of abstraction. In the current version of iNNspector, navigation over levels of abstraction feels like browsing the z-axis while zooming and panning cover the x-y-plane. In our user study (see section 4.5.2) we observed the navigation over levels of abstraction to be intuitive to the users.

4.4.3 Developer Workflow

The iNNspector system is designed to be easily integrated into the model-building workflow. In the following, we will describe the steps needed to bring an experiment into the iNNspector system. iNNspector provides a Python package, consisting of three main components: (1) the custom model checkpoint, providing the logging mechanisms, (2) several pre-defined data generators, implementing convenience functions to retrieve

the number of classes and a subset of the evaluation dataset, and (3) a database connector, providing methods to store and load the model's metadata to and from the model database.

Creating a Model — The model creation process is similar to the one in ordinary TensorFlow / Keras. The only difference is that the model has to be annotated with an iNNspector header function (see paragraph 4.4.4.1.1, "iNNspector Keras Logs"). The function has to be called before training the model and returns a dictionary containing the model's meta information.

Training the Model — For training, an instance of the customized iNNspector checkpoint (see paragraph 4.4.4.1.2, "iNNspector Keras Logs") has to be passed as a callback to the *fit* method of the model. The checkpoint receives a reference to the dictionary created by the iNNspector header function and extends it with information on the location of the checkpoint files and the graph file.

Storing the Metadata to the Model Database — After training, the model's metadata has to be appended to the model database using the database connector.

Creating New Model Iterations — When refining the initial architecture to generate a new model iteration, the user has to create a new iNNspector header, denoting the new model as a child of the previous one. This can be done manually or by using the *Branch Model* tool, which automatically generates a new iNNspector header with the selected model as parent.

Inspecting Models in the User-Interface — After starting the iNNspector back- and frontend, the stored models are automatically loaded into the system.

4.4.4 System Architecture & Implementation

The iNNspector system is driven by custom logging mechanisms and a backend providing access to the data accumulated during training. In the following, we provide a short overview over these components to convey an impression on how the system works and which steps it takes to integrate it into the model building workflow.

4.4.4.1 iNNspector Keras Logs

We provide two custom components to make ordinary TensorFlow / Keras available in the iNNspector system. The first one is the *iNNspector* header, which provides metadata about a model, and the second one is the *iNNspector Keras Log*, which modifies the default checkpoints to also include a subset of the testing dataset together with the

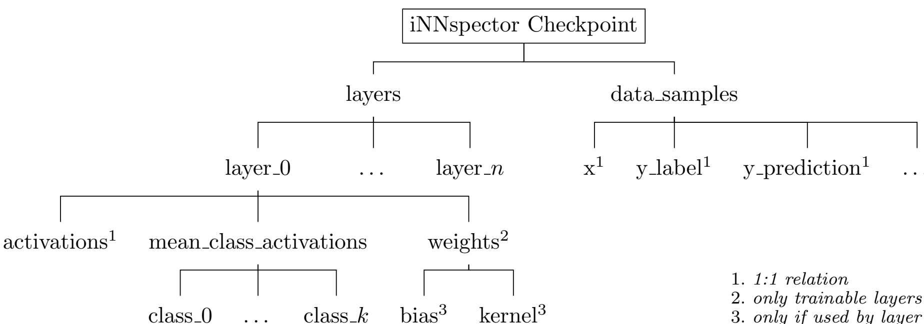


Figure 4.13: HDF5-structure of iNNspector checkpoint files. We store weights and activations layerwise, while dataset samples, their labels (if available), and their predictions are stored globally per checkpoint.

corresponding activations for each layer. In the following, we will describe how those two components work together to bring models into the iNNspector system.

iNNspector Header — The iNNspector header is a small code block in the model definition file encoding the model's meta-information in a key-value structure, such as its name and parent-child relationships. Listing 4.1 shows an exemplary function to generate the header.

During model training, the iNNspector header is used by the iNNspector logging mechanism to create a model database. Besides meta information, the database encodes information on the location of graph files and checkpoints, values of performance metrics, and other high-level statistics, such as the number of trainable parameters, memory consumption, and creation time of the model. To reduce manual effort, the *Branch Model* tool automatically generates a new iNNspector header with the selected model as parent.

iNNspector Checkpoint — The iNNspector checkpoint extends the default Keras checkpoint by dataset samples and sample activations. For that, we re-structure the HDF5 file according to the hierarchy diagrammed in figure 4.13. Dataset samples ($\sim x$), dataset labels ($\sim$ *y_label*), and model predictions ($\sim$ *y_prediction*) are stored globally per checkpoint in the *data_samples* group, since they do not change over layers. In

```
1 def make_innspector_header():
2     return {
3         "id": "0999a51f-5114-471a-8322-9fa40ae60387",
4         "name": "ts_mnist_dense_skip-connections",
5         "label": "MNIST Dense (Skip Connections)",
6         "parents": ["bab98735-d119-433d-a893-08232533973e"]
7     }
```

Listing 4.1: Function to generate the iNNspector header encoding a model's meta information.

contrast, weights and activations have to be stored layerwise. Therefore, the *layers* group contains an entry for every layer of the model, referenced by the Keras layer name. Since activations and predictions are computed per dataset sample, inputs, labels, predictions, and activations share a one-to-one relationship, reflecting in equal size of their first matrix dimension. Furthermore, each layer stores pre-computed averaged activations for the individual classes of the dataset (∼ *mean_class_activations*). This avoids that the backend has to repeatedly load and aggregate all activations, since this data is frequently requested by the iNNspector frontend. Layers with trainable variables also have a *weights* group, containing the values of the layer's *kernel* or *bias*, respectively. It should be noted that storing pre-computed activations might not be viable for huge models or models featuring convolutions with large output sizes due to storage size issues. In this case, we would have to fall back to loading and executing the saved model instance on the fly with the drawback of higher response times. To give an estimate: for our three use cases, we trained eight image classification models and ten variational auto-encoders with different architectures on the MNIST dataset. Each model had eleven iNNspector checkpoints saved over the training, resulting in an overall size of ≈ 70 GB.

4.4.4.2 iNNspector Backend

The iNNspector backend exposes the data stored in the iNNspector system over different REST API routes for access from the iNNspector frontend. It constantly monitors the contents of the logging directory and reloads the model database upon changes. The API endpoints can be grouped into the following categories:

Model IDs A single `GET` -endpoint returning the IDs of all models in the database.

Model Info Different `GET` -endpoints taking the model ID as a parameter and returning information about the model, such as architecture graph, performance metrics, or a catalog of checkpoints. All this information is stored in the model database.

Checkpoints Different `POST` -endpoints taking the model ID and a JSON body as parameter and returning (transformed) values stored in– and extracted from checkpoints, such as weights, activations, or interestingness values.

The API is built using FastAPI[1] and provides detailed OpenAPI[2] documentation, accessible under the backend URL. To speed up repeated queries with similar parameters, we cache results in Redis for later re-use.

Data Transformation — High-dimensional data **(D3)** necessarily has to undergo task-specific processing **(I3)** to create human-interpretable representations **(M6)**. Therefore,

[1] `https://fastapi.tiangolo.com/`
[2] `https://swagger.io/specification/`

all backend routes returning high-dimensional data take an additional **transform specification**, allowing to express arbitrary transformations that should be applied to the data. The transform specification is defined in a grammar which allows specifying a list of operations that are subsequently applied. Those operations map functionalities of Pandas and Numpy into our grammar. Furthermore, our grammar provides *production rules*, allowing, e.g., to branch and merge data columns.

By using transformations to filter and aggregate the requested data aggressively, we can mitigate the issues with the huge size of some high-dimensional data entities, facilitating the smooth exchange of data between backend and frontend over HTTP requests. The transformations allow us to trade memory size for computational effort; combining them with caching allows us to utilize both advantages, leading to quicker overall response times.

4.4.4.3 iNNspector Frontend

Interface and functionalities of the frontend are described exhaustively in section 4.4.1. The frontend is built using React[3] and TypeScript[4]. The modularity enabled by the React framework and the strong type system enforced by TypeScript allow new developers to quickly find their way around the code, supporting our goal of system extensibility.

4.5 Evaluation

Our evaluation of iNNspector is two-fold: First, we present three real-world use cases that demonstrate how **iNNspector** can be used to systematically debug deep learning experiments. Second, we perform an expert user study with deep learning experts to evaluate the usability of **iNNspector** and to gather feedback on the usefulness of the system.

4.5.1 Use Cases

We show the applicability of our approach based on three use cases, which we derived from our model developer interviews and described in section 4.2.3.2. We selected these three use cases, since, during the interviews, the developers emphasized their importance. All mechanisms proposed in section 4.3.3 and their corresponding implementations described in section 4.4 will be utilized to solve the use cases, providing

[3] `https://reactjs.org/`
[4] `https://www.typescriptlang.org/`

an impression on how their integration in the everyday model building, model diagnosis, and model refinement workflow enables systematic analysis and debugging of the connection between model behavior and its architectural parts. Use case 1 is relatively universal, demonstrating how the system facilitates the everyday model building and –refinement workflow. Subsequently, use cases 2 and 3 go into increasing detail on a specific debugging task. We utilize these use cases in our user study, as described in section 4.5.2.

4.5.1.1 Use Case 1: Model Assessment, –Comparison, and –Refinement

A model developer wants to build a model for image classification, which should eventually run in real-time on an embedded device. The hardware has low computational power and limited storage size. Therefore, execution time and model save size must be minimized while maximizing the classification performance. Based on his prior experiences in deep learning, the developer creates an initial model architecture. During training, the model already shows promising results, reaching an accuracy of 88% after 20 epochs. However, he wants to refine architecture and hyperparameters further to boost the model's performance while keeping the hardware requirements constant. In an iterative approach, the developer creates multiple model variations, experimenting with 2D convolutions, network structures, and layer sizes. Thereby, he uses the iNNspector logging mechanism to track and debug his experiments in the iNNspector system. Using L3, the *tree of models* indicates the variations between the models in the abstract model architecture representation while also confirming the change in trainable parameters between model variants. By applying the *performance metrics* tool to the group of trained models, he can evaluate and compare their classification accuracy. Since computational complexity matters in the real-time scenario, the *runtime statistics* tool provides a good estimate of the expected model execution time. Respectively, the *model save size* tool gives an overview of the model size. Comparing accuracy, save size, and execution times, two models seem promising, as highlighted in figure 4.14. The developer adds single-model *performance metrics* widgets for those two models individually. To estimate their tendency to overfit, he merges the widgets in the group view to compare training– and validation loss, showing that one model seems to better generalize than the other. Outgoing from this superior model variant, the developer wants to further tweak the performance by assessing the configuration of its layers and inspecting each layer's value distribution. He navigates to L2, exposing more architectural details and providing access to the model's variables. He creates widgets visualizing convolutional and dense kernels, enabling him to compare the individual data distributions. In the second layer, the variance increases significantly over the training epochs, while the third layer shows many values staying close to zero, indicating a bottleneck in layer two, which limits the information flow. With this insight, the developer has a clear leverage point for further

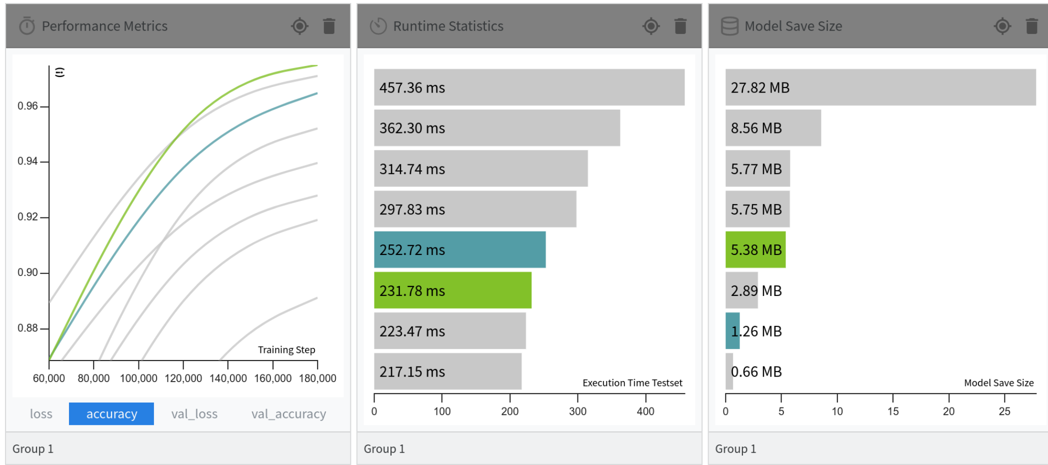


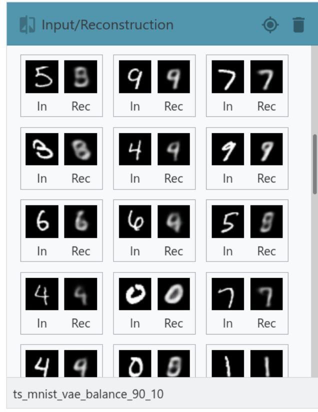
Figure 4.14: iNNspector widgets to assess model performance over an experiment. The *Performance Metrics* widget shows the accuracy of each model, the *Runtime Statistics* widget shows their expected execution time, and the *Model Save Size* widget shows the model's size on disk.

model tweaking. He creates a new generation of models with improved performance by re-balancing the capacity between layers two and three.

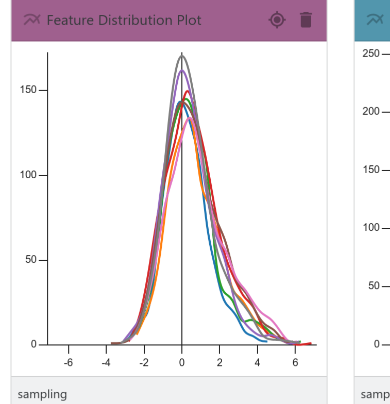
4.5.1.2 Use Case 2: Balancing Losses

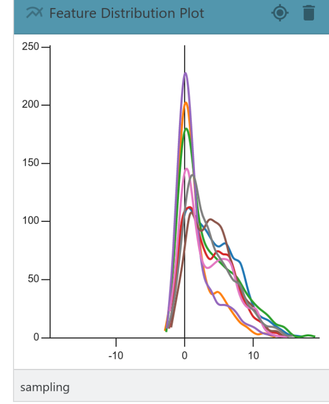
A developer wants to build a variational auto-encoder (VAE) [213] for image generation. Constraining the latent distribution to resemble a normal distribution ensures that when sampling the latent vector from a normal distribution, valid decoder output is generated. The developer uses the Kullback–Leibler (KL) divergence to calculate the difference between the actual distribution in the latent space and the desired normal distribution. To optimize both reconstruction results and shape of the latent distribution, he must precisely balance reconstruction loss and KL divergence against each other. However, due to this parameter being dependent on dataset, model, and human perception, there is no analytical solution to this problem. Therefore, the developer creates multiple variations of the network, altering the loss balancing factor between models. The customized Keras model overrides the `train_step` and `test_step` functions to also return KL– and reconstruction loss, which are, thereby, automatically tracked by the iNNspector logging mechansism.

While the reconstruction loss can be visually evaluated by simply displaying the reconstructed images using the *input/reconstruction* tool shown in figure 4.15a, the KL loss is a rather abstract factor. To assess it and examine the value distribution in the latent space, the developer descends into L2, enabling him to visualize how the *overall* distribution of activations l in the latent layer developed over the training epochs. Using automated interestingness measures, he finds that the model with a 25 : 75 balance between



(a) Image reconstructions for different dataset samples.





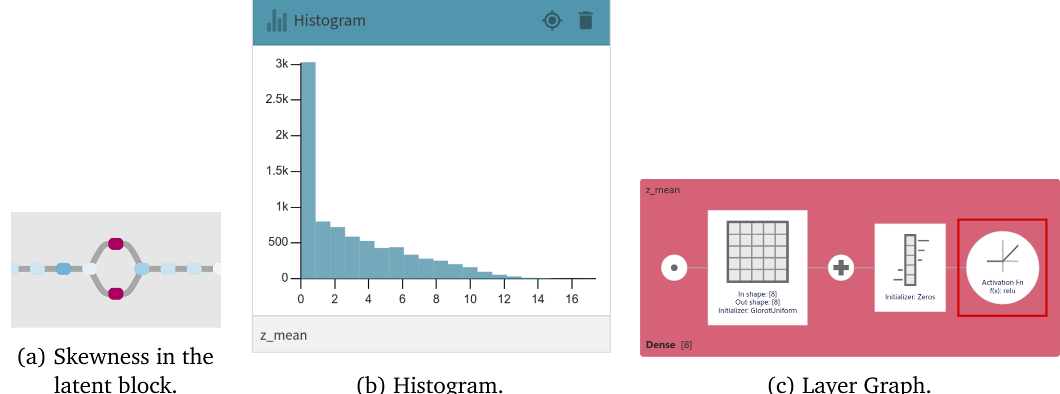
(b) Feature distribution (∼ probability density function) for each dimension in the latent variable of two models.

Figure 4.15: iNNspector widgets to balance losses in a VAE. (a) The *input/reconstruction* widget shows the input images and their reconstructions for a subset of the test dataset. (b) The *feature distribution* widget shows the shape of the latent distribution, suggesting the left model to be better suited for sampling.

reconstruction– and KL loss has a much lower variance of $\mathrm{Var}\left(l_{25:75}\right) = 2.02$ than the model with a 90 : 10 balance, where the variance is $\mathrm{Var}\left(l_{90:10}\right) = 12.38$. Since, in VAEs, uniformity of each dimension in the latent variable is important, the developer uses the *feature distribution plot* to plot the probability density function for each dimension individually, depicted in 4.15b. It shows that the model with more weight on the KL loss has a more uniform distribution in all latent dimensions. For the following iterations of the model development and refinement process, the inspection results enable the model developer to make well-founded decisions over the loss balance.

4.5.1.3 Use Case 3: Debugging Distributions

Extending use case 2, while weighting reconstruction and KL loss, the developer seems not to be able to find a satisfying balance. Instead, either the reconstructed image or the latent distribution deviates significantly from the desired results. The strong skew of the the activations in the latent space, visualized by the widgets in figure 4.15b, prompts further investigation. Using the *Activation Skew* interestingness measure on L3, he notices even stronger skew in two layers of the latent block, depicted in figure 4.16a. To trace down the detected abnormality to its root cause, the developer descends to L2 and closely investigates the respective layers, namely the `z_mean` and `z_log_var` layers, which determine the parameters of the normal distribution. Applying the *Histogram* tool confirms this observation, as shown in figure 4.16b; there are only positive activations on that layer. Upon closer inspection of the layer's inner elements, the developer identifies the reasons for that behavior: due to a careless mistake when defining the



(a) Skewness in the latent block.

(b) Histogram.

(c) Layer Graph.

Figure 4.16: Widgets to investigate the distribution of activations in the latent block. (a) Interestingngess annotations points the developer to the skewness of the activations in the latent block. (b) The *Histogram* widget provides a detailed view on the distribution of activations in the latent layer. (c) The *Layer Graph* reveals the erroneous activation function, leading to the skewed distribution.

Dense layer in Keras, the layer's output is passed through a *ReLu* activation function depicted in figure 4.16c, resulting in all negative values being cut off. The developer creates a new model generation using the *Branch Model* tool and removes the suspicious activation function. Re-training the new model proves the refinement's success.

4.5.2 Expert User Study

To evaluate the iNNspector system, we conduct a user study with four participants. The prior objectives of the evaluation study are to assess (1) whether the system does fulfill its claim of being a universal tool for model debugging, (2) the usability and effectiveness of the system, and (3) identify open challenges and future work.

4.5.2.1 Study Setup

In this section, we describe our study setup, guided by the methodology proposed by Sperrle et al. [214]. First, we explain our study procedure for each session; second, we describe the participants of our study; and lastly, we discuss the data used in our pair analytics sessions.

Study Procedure — We divide each study session into four blocks. Each session takes approximately two hours and is recorded for later transcription and qualitative evaluation.

Current Workflow (≈ 5 min): We start by capturing personal information, such as the amount of experience our participants have with deep learning. We continue with questions on the participant's usual development workflow, including questions on experiment tracking, model assessment, and potential debugging routines. This block is closed by open-ended questions on deficiencies of current debugging tools and wishes towards a system for universal deep model inspection.

Formal Framework (≈ 10 min): By reference to table 4.1 and figure 4.3, we briefly describe the parts of our conceptual framework being most relevant for the implementation of a debugging system, namely the data categories (4.3.1) and the mechanisms to make this data accessible (4.3.3). Following, we capture feedback on the framework in a semi-structured interview, covering its completeness and anticipated applicability.

iNNspector System & Use Cases (≈ 45 min): This is the central part of the interview, taking up about one hour of the overall two-hour interview slot. In a guided exploration phase, we explain the iNNspector user interface to the participant, similar to the application walkthrough in section 4.4.1. When the participants confirm they feel confident with the interface, they are successively asked to solve use cases 1 to 3 described in section 4.5.1 in a pair-analytics [215] session. The participants receive a handout explaining the setting of each use case and defining its corresponding analysis tasks, which we also summarize verbally. The participants are supposed to solve the use cases mainly on their own; however, when they articulate that they are looking for a particular tool or have questions on either task or system, we jump in to help. To capture impressions and feedback on the system, we encourage the participants to speak out loud about their reasoning during this process.

Questionnaire and Feedback (≈ 15 min): Finally, we collect quantitative and qualitative user feedback on the system. Based on a customized questionnaire, inspired by the NASA Task Load Index [216] and the System Usability Scale [217], we evaluate iNNspector's usability, usefulness, task load, and effects on the debugging behavior. Furthermore, the questionnaire captures the usefulness of specific system components, such as the different levels of abstraction or the toolbox. The interview finishes with open-ended questions and broader feedback on the system, including missing components, ideas for future extensions, and general thoughts on the system.

Participants — All four participants (P1, P2, P3, and P4) have an educational background in information science on masters level. P1, P2, and P4 have specialized in deep learning and show multiple years of experience with deep learning development in both research and industry. P3 has had prior experiences in deep learning as part of

his bachelors and master's degrees. P1 and P4 were already part of the requirements study, while P2 and P3 were only part of the evaluation study. Neither the study leader nor the participants are native English speakers; nevertheless, the study sessions are conducted in English to achieve the most accurate transcription and citation.

Data — The models available in the iNNspector system are pre-trained on the MNIST dataset. For each use case, a selection of models supporting the use case is presented to the user. The models simulate a representative real-world model building and refinement workflow, with model iterations implementing architectural changes based on observations from past generations. For use case 1, the models are classifiers, while for use cases 2 to 4 the models are variational auto-encoders.

4.5.2.2 Study Results

While the participants showed different exploration strategies, they all reached the analysis goals of use cases 1 and 2 in similar times (P1: 40 min, P2: 36 min, P3: 22 min, and P4: 30 min). P3 was the fastest participant; however, also being the least experienced, he did not reach an analysis depth comparable to the other participants, despite in the end drawing the same conclusions. Use case 3 was introduced as an optional, open-ended task, where we only added a new model generation with the suspicious activation function removed and asked the participant to determine (1) possible issues with the previous generation and (2) how the new generation changed and why this resolves the issue. Here, the participants focussed on comparing the architectures and, subsequently, found the difference in the latent block. However, they did not recognize a major improvement, which might be due to relatively blurry reconstruction results of both model variants, caused by the limited capacity of the latent variable ($\text{dim} = 10$).

In the following, we summarize the feedback of our participants. Due to the limited number of participants, we present quantitative and qualitative results together. Before facing our participants with the iNNspector system, we collect feedback on our conceptual framework. Particularly, we introduce the data categories (section 4.3.1) and global mechanisms (section 4.3.3) and collect feedback on the framework's correctness, completeness, and usefulness.

Data — In general, our participants confirmed the data categories to make sense and match their mental model about the data arising in machine learning experiments (P1, P2, P3, P4). P1 was missing a data category for meta-parameters about the experiment, such as batch size or optimization parameters. P2 had similar concerns, but readily proposed an integration of those parameters into our categorization as *constant scalar* data, which perfectly complies with our model. We updated the categorization accordingly. P2

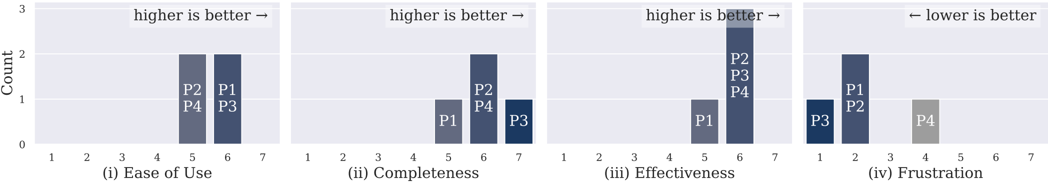


Figure 4.17: Quantitative study results on the **usability** of the iNNspector system. The bars show the count of participants rating the system with the respective score.

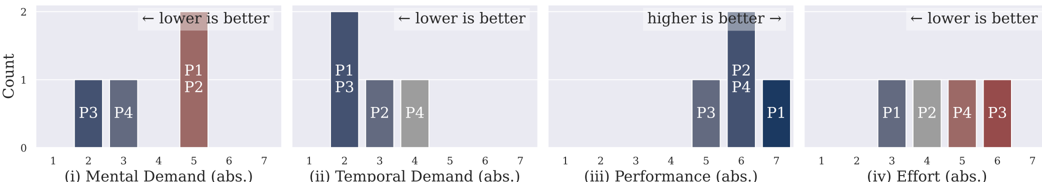
also mentioned that he sees functions (D4) as part of the structural architecture graph (D1.2) since they "define the geometry of the graph." We added a respective pointer to table 4.1.

Mechanisms — When introducing the mechanisms, our participants instinctively started to compare the framework to state-of-the-art tools for model debugging. E.g., P4 identified a discrepancy between usefulness and convenience in existing tools: "either you need to self-make your tools, which are very low level [...], or you can use TensorBoard but they have only very general statistical evaluations." Therefore, he valued the framework as an "improvement over existing tools [...], if applied [implemented] correctly." With regards to the proposed data representation (M6), P2 stated that "scalar values and images can be visualized in existing tools, but that's mostly it.". Also, he mentioned to not use the graph view in TensorBoard very often, since it yields "not much new information." In contrast, our framework proposes the entities of the structural backbone (M2) as a navigational component and a hook to access underlying data. Overall, our participants agreed that the mechanisms were reasonable (P1, P2, P3, P4) and expressed their anticipation for the system implementation.

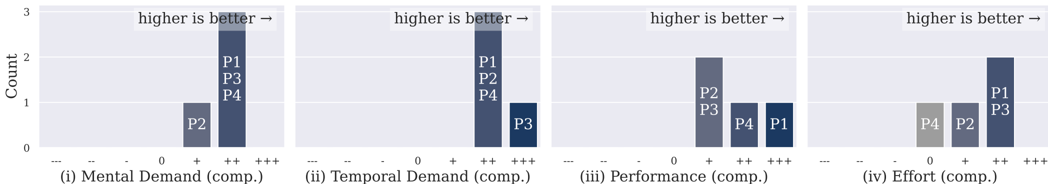
Gathering feedback on the system is divided into three phases: first, we introduce our participants to the iNNspector system in a joint exploration phase. We describe the interface similar to section 4.4.1 while the participant already has full control over the system to discover and explore its functionalities. In the second phase, the user solves the use cases 1 to 3 described in section 4.5.1. Finally, we gather feedback via a questionnaire containing both quantitative and qualitative questions. In the following, we summarize the feedback structured by our main evaluation goals *usability*, *usefulnes*, and the *claim of the system to be a generalizing tool for systematic model debugging.*

Usability — Usability is captured in terms of *ease of use*, *completeness*, *effectiveness*, and *frustration*. Figure 4.17 shows the quantitative results of the evaluation, which are substantiated by qualitative feedback in the following.

Overall, the system was described as "intuitive" (P1, P3, P4) and "very usable" (P2). Particularly, functionalities like the navigation over levels of abstraction, filters, and widget groups were intuitively used by all our participants. In contrast, some participants needed a quick reminder that the class selector panel could help solve use case 2. The



(a) **Absolute** usefulness rating, i.e., without considering the complexity of the tasks or how it would be solved using traditional tools.



(b) Relative usefulness rating, comparing to the **usual workflow as a baseline**. "– (– –)" means a (strong) negative and "+(++)" a (strong) positive influence of the iNNspector system.

Figure 4.18: Quantitative results on the **usefulness** of the system. While the mental demand and effort are rated high on the absolute scala, the system is rated as a great improvement in comparison to the usual workflow.

two major points of critique were that the system is quite complex and would need training to fully exploit all functionalities (P1, P2, P4) and that the tool icons and –descriptions could be improved (P1, P3, P4). In general, the participants were pleased with the completeness of the system. The primary point of critique were missing tools, with a confusion matrix being the most frequently requested (P1, P2, P4). As a response, we added the tool after the study. The effectiveness of the tool was rated between okay and good without further comments. The frustration factor was low for most participants (P1, P2, P3), despite a memory leakage bug occurring for P2 causing significant lag. However, P2 stated that "[the bug] was not a big deal." In the meantime, the error has been fixed. P1 told the frustration to be "actually very low"; however, he also experienced a bug causing the class selector to malfunction: "[if there wasn't the bug], I would put it at 1, as [the system] works quite nice." P4 stated medium frustration, caused by a lack of tools to solve use case 2. Particularly, he was missing a confusion matrix tool and linking and brushing between misclassified samples across different models.

Usefulness — Usefulness is captured in terms of *mental demand*, *temporal demand*, *performance*, and *effort*. Each question has two parts, with the first one asking absolute values (figure 4.18a) and the second one comparing to the usual workflow of our participants as a baseline (figure 4.18b). In the following, we will report both the quantitative and qualitative feedback on the system's usefullness.

Despite the mental demand of use cases 1 to 3 being rated relatively high on the absolute scale (P1, P2), all participants aggreed that the system significantly simplified the tasks

and "helped a lot so that the mental demand was not very high." However, P2 mentioned concerns about mental overload when using the system: "it's more stuff at once; for example, in a Jupyter notebook or in my normal workflow it's more sequential." Our participants agreed on the system strongly decreasing the temporal demand, rating the time gain high (P1, P2, P4) to very high (P3). P2 noted that "doing all of [the debugging in the use cases] would definitely take some time, configurations, and stuff." P1 particularly mentioned that the system makes it "much easier to compare different architectures." All participants were satisfied with the performance they achieved when solving the use cases. In comparison, P2 at first expected similar performance with traditional tools but on second thought revised his opinion: "with the latent distributions it would have taken way longer and would not be that easy." Generally, the absolute effort to solve the tasks was rated mid (P1, P2) to high (P3, P4). P4 ascribed this mainly to being "new to the system." However, in comparison, the system still was rated as an improvement (P2, P3, P4). Only P4 was sceptical: "[the system did not reduce the effort] in comparison with statistical evaluation."

Adequacy for Systematic Model Debugging — Concluding our questionnaire, we capture general impressions of the system based on the following questions.

(i) *How much has the system increased or simplified your access to the data arising in machine learning experiments?*

All our participants agreed that iNNspector simplifies access to data greatly (P1, P2, P3) to very greatly (P4). P4 confirmed the coverage: "you can look at everything you want." Overall, during the exploration– and use case phases of our sessions, none of the participants requested data that was not available in the system; critique was only mentioned about the representation of data, i.e., our participants wanted to customize tools and widgets according to their needs.

(ii) *How much time would iNNspector save you in your everyday model development and debugging workflow?*

The time saving was rated between neutral (P2) over good (P1, P4) to very high (P3). P2 expressed his uncertainty: "I can not really say yet. I would have to use it a bit more." Since the models were pre-trained preliminary to the study due to time constraints, our participants gave to consideration that "this depends on how much time you need to set up the model[s] like this" (P3). After explaining the logging mechanism described in section 4.4.4.1, P3 added that "this sounds like it is really easy; so I think it would save really much time."

(iii) *In your opinion, to what extent does iNNspector fulfill the claim of being a universal tool for systematic deep model debugging?*

The universality of the system was rated from okay (P1) over good (P2, P3) to very high (P4). Here, customizability and expandability were the decisive points

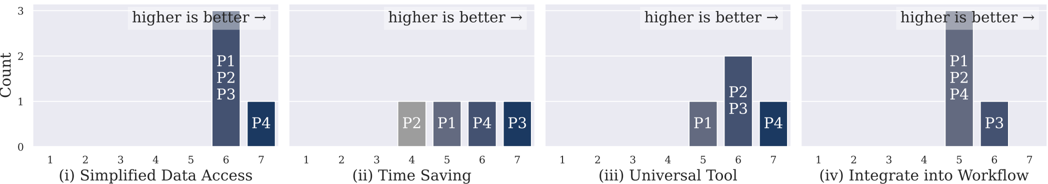


Figure 4.19: Quantitative results on the **adequacy** of the iNNspector system for **systematic model debugging**. Particularly the system's ability to simplify access to the data arising in deep learning experiments was rated high.

for all participants. However, P1 added that the system "already has a lot to offer." P4 justified his distinct vote: "I think I couldn't come up with a use case that I couldn't solve with the system."

(iv) *How likely are you to deploy iNNspector in your everyday model development and debugging process?*

P1, P2, and P4 stated they would probably integrate the system into their everyday workflow. P3 not being confronted with deep leraning on a regular basis reasoned "[he] would like to use this if [he] would develop machine learning models." Again, the simplicity of the logging mechanism was identified as crucial: "if I could run it easily with my models, I most likely would use it."

4.6 Discussion

Summarizing our work, we identify concrete recommendations guiding the design of real-world systems for the systematic debugging of machine learning experiments. Furthermore, we discuss current limitations of our approach and how they will be addressed in future work.

4.6.1 Take-Home Messages

This section summarizes the major take-home messages of our approach and distills general recommendations for the design of systems for systematic network debugging.

(IN1) Consistent Logging of Appropriate Data — All relevant data generated in machine learning experiments should be logged for later use in the debugging stage. Model evolution should be actively tracked by encoding parent-child relationships in the model metadata, allowing later reconstructions of the experiment's progression. Logging sections of the training and testing dataset is crucial in two ways: firstly, single samples and their model output can be inspected in the debugging stage, e.g., to investigate

miss-classifications. Secondly, the activations of a model only exist under input data. Therefore, the activations should be computed and stored for each model checkpoint and data sample, allowing the inspection of activation distributions and single neurons.

Notably, the so-logged data can grow to a significant size. However, we argue that the vast majority of physical storage requirements trace back to checkpoints, which are in most cases logged anyway to restore a particular model state. If the amount of logged data exceeds a feasible storage size, pre-aggregations before storing the data can significantly reduce its size. Since this comes at the cost of reduced flexibility in the debugging stage, the advantages and disadvantages of pre-aggregations should be carefully weighed. For architectures producing particularly large activation data (e.g., convolutional networks on large image sizes), the activations can be computed on the fly with the downside of slower query times.

(IN2) Modular Approach for Systematic Debugging — Often, there is no ideal default representation of the data to inspect. Our proposed framework tackles this challenge by structuring the dimensions that influence and determine the data representation, maximizing flexibility while simultaneously resolving mutual dependencies. The presented system shows how different visualization and interaction techniques can work together to combine these dimensions in a reasonable way.

(IN3) Global Mechanisms — The global mechanisms to navigate the data space elaborated in section 4.3.3 provide entry points for implementing a system for the systematic debugging of deep learning models. While not all proposed mechanisms have to be strictly followed to make such a system work, they highlight the challenges that must be considered and suggest potential solutions. The iNNspector system demonstrates, how the mechanisms can be instantiated into a real-world application, giving the developer access to the vast data space of machine learning experiments.

4.6.2 Limitations and Future Work

In more complex experiments, both the **tree of models** and the **architecture graph** might rapidly grow to an unmanagable size, leading to information overload, as also noted by several of our study participants. To mitigate this problem we will include mechanisms for semantic block detection in future versions of the system, leveraging common properties of the iterative model refinement process and architecture graphs. Particularly, model variations often share common architecture or hyper-parameter configurations, e.g., we could visually group models that only contain changes in one parameter, such as the batch size. Likewise, architecture graphs frequently contain repetitions of similar structures, e.g., stacked residual blocks [212], or common semantic

blocks, e.g., encoder/decoder structures [18]. Again, such blocks could be visually abstracted, significantly reducing the complexity of advanced model architectures.

Extensibility — The iNNspector system is designed to be easily extendable by own tools and widgets. To fully determine tool and corresponding widget, the following aspects have to be defined:

Tool Defintion — The appearance of the tool, including icon, name, description, and category. Furthermore, the UoA type to which the tool is applicable has to be specified.

Data Source — The kind of data to query from the backend, i.e., the API endpoint together with its corresponding paramters. This includes model id and the kind of data, but also the transformation specification.

Visual Appearance — Finally, the appearence of the widget, including visualizations and interactions, has to be configured in a standardized way.

While at the moment, this still affords programming skills in React and TypeScript, future versions of the system will support specifying tools as external plugins. Particularly, tool definition and data source can easily be defined in a standardized data-serialization format, such as YAML[5] or JSON[6]. To also support such definition for the visual appearance of the widget, we rely on the Vega-Lite grammar of interactive graphics [218], allowing to define complex interactive visualizations in plain JSON.

Specialized Default Tools — Complementary to the extensibility of the system, we will provide additional, specialized tools for common debugging tasks. Particularly, as identified in our conceptual framework (see section 4.3.2.1, "Design Dimensions for Data Representation and Interaction"), we consider formal verification and correctness checking as important debugging tasks. Therefore, as a first step towards formal verification, e.g., through SMT [219], we will provide tools to investigate the decision boundaries of neural networks, e.g., through counterfactuals [220] and to check its robustness, e.g., through adversarial examples [221, 222].

Transferability — Its flexible design, the described extensibility, and the formal guidelines provided by the framework make iNNspector generalize well to various types of data, models, and tasks. However, particular application domains might afford extensions going beyond defining new widgets and tools. E.g., to debug models for deep reinforcement learning, closely-integrated tools to visualize state space and agent policies might be beneficial [223, 224]. Another example are language transformers [18], where a tight integration of attention visualizations into the structural backbone seems useful.

[5] https://yaml.org/
[6] https://www.json.org/json-en.html

Explainability — Future versions of iNNspector are connected to the explAIner system for explainable AI and interactive machine learning presented in chapter 3, providing access to a variety of model explainability techniques. A direct integration of the explAIner backend into the iNNspector system facilitates local, model-agnostic debugging. This enriches the iNNspector toolbox with various state-of-the-art XAI techniques, such as LIME [20], LRP [23], or Integrated Gradients [32].

Comparative Analytics — Model comparison is an essential task for deep learning developers, which was confirmed in our requirements– and evaluation interviews. While iNNspector already supports comparison of scalar– and high-dimensional data through multi-model– and side-by-side widgets, a dedicated tool for architecture comparison can enhance the analysis of structural data. For this, in future versions, we generate node embeddings for the architectural blocks of a model, allowing to compute a bi-directional mapping between two model graphs. This mapping is used to align the graphs and visualize them side-by-side, highlighting commonalities and differences in the two architectures.

4.7 Conclusion

With this work, we facilitate the systematic debugging of deep learning experiments.

As a formal foundation for implementing systems enabling this, we structure the design space of such systems and compile it into a **conceptual framework**. Particularly, the framework (1) categorizes the data arising in machine learning experiments, (2) captures the design dimensions that have to be considered when building such a system, and (3) defines concrete mechanisms to incorporate data and design dimensions into a comprehensive system for systematic debugging. The considerations that went into the system are derived from (1) open challenges in the field and (2) requirement interviews with real-world deep learning developers.

Our framework proposes modular UI elements, called *debugging components*, which can be instantiated in a toolbox-like approach. The user determines the appearance of a debugging component through prioritizing characteristics of the available design dimensions. E.g., the user can choose between an *assessment* or *comparison task*, or a *visual* or *verbal data representation*. Debugging components can be attached to different *units of analysis* in the model architecture. We propose to make the model architectures explorable on different *levels of abstraction*, ranging from a multi-model view down to single weights and neurons.

We transfer our conceptual work into a ready-to-use **system implementation** called ***iNNspector***. Analogously to the framework, iNNspector lets the user explore models

and their architectures over multiple levels of abstraction. Different tools are available in a toolbox, which can be applied to units of analysis in the architecture representation. The application of a tool results in creating a widget displaying the data. The widget's appearance is determined by the tools specification and the UoA type it is applied on, automatically resolving dependencies between design dimensions. Various additional functionalities, such as interestingness measures, localization of UoA, or class selection, enrich the system on a global scale. Notably, iNNspector goes beyond demonstrating the feasibility of our framework; upon publication of this work, it will be released as open-source software and, hopefully, find its way into the everyday model building and –debugging workflow of deep learning developers.

The evaluation of the iNNspector is twofold. First, we present three use cases demonstrating how it can be used to debug a variety of problems occurring in real-world model-building scenarios. Second, we conduct a user study with three deep learning developers and a data scientist to evaluate the system's usability, usefulness, and versatility.

We argue that a global understanding of models and their evolution can only be achieved by systematically debugging, building the foundation for a well-informed diagnosis, verification, and refinement process. All data arising in machine learning experiments are relevant for the debugging stage and should be logged and made explorable. iNNspector enables this kind of data tracking and systematic debugging.

5 generAItor — Explainable Language Model Predictions

Large language models are widely deployed in various downstream tasks, e.g., auto-completion, aided writing, or chat-based text generation. However, the considered output candidates of the underlying search algorithm are under-explored and under-explained. We tackle this shortcoming by proposing a *tree-in-the-loop* approach, where a visual representation of the beam search tree is the central component for analyzing, explaining, and adapting the generated outputs. To support these tasks, in this chapter, we present generAItor, a visual analytics technique augmenting the central beam search tree with various task-specific widgets, providing targeted visualizations and interaction possibilities. Our approach allows interactions on multiple levels and offers an iterative pipeline that encompasses generating, exploring, and comparing output candidates, as well as fine-tuning the model based on adapted data. Our case study shows that our tool generates new insights into gender bias analysis beyond state-of-the-art template-based methods. Additionally, we demonstrate the applicability of our approach in a qualitative user study. Also, we quantitatively evaluate the adaptability of the model to few samples, as occurring in text-generation use cases. Finally, we apply our approach to several state-of-the-art linguistic challenges concerning the prompting of LLMs.

The chapter is based on the following publications. For a detailed contribution clarification, refer to section 1.4.

[3] T. Spinner, R. Kehlbeck, R. Sevastjanova, T. Stähle, D. A. Keim, O. Deussen, and M. El-Assady. “generAItor: Tree-in-the-Loop Text Generation for Language Model Explainability and Adaptation”. In: *ACM Transactions on Interactive Intelligent Systems* 14.2 (2024).

[4] T. Spinner, R. Kehlbeck, R. Sevastjanova, T. Stähle, D. A. Keim, O. Deussen, A. Spitz, and M. El-Assady. “Revealing the Unwritten: Visual Investigation of Beam Search Trees to Address Language Model Prompting Challenges”. 2023. arXiv: 2310.11252.

5.1 Introduction

Recently, large language models have gained increased popularity, especially in the field of natural language generation. At the latest, with the introduction of ChatGPT[1], LLMs have been made accessible to a wider, more general audience. Despite their growing recognition and notable accomplishments, LLMs still face several limitations. Common failures, even for state-of-the-art models, are repetitive content, the lack of factual accuracy, often referred to as hallucination [225], and biases [226]. However, the perceived high quality of the outputs makes it difficult to identify such errors without a deeper understanding of the model's decision-making process. Particularly, the chat interface of ChatGPT and other chat- or completion-based approaches omit important information on uncertainties or viable alternatives from the users. While text-based interfaces may fulfill the needs for a broad, general audience, **interested non-experts** and **linguistic experts** require more in-depth insights and control.

We identify three primary shortcomings in the current state-of-the-art for interacting with LLMs: lack of **explainability**, **comparability**, and **adaptability**. Explainability refers to understanding of the model's decision process, including the way a language model predicts its output, its sampling strategy, and the probabilities of these outputs. For example, explanations of a prediction's certainty can provide the user a hint on possible hallucinations. Comparability, i.e., a simple yet effective comparison of multiple generated outputs, can enable the user to assess more specific nuances in the model's predictions. This is particularly relevant for linguistic experts. For instance, by adapting prompts with typical names from varying ethnic groups and comparing the predictions, the user can assess the model's biases, if present. And lastly, adaptability is relevant when the generated output is not satisfactory. In these situations, the user should be able to edit problematic parts; e.g., by correcting made-up facts and making these changes permanent; e.g., by fine-tuning the model.

To address the identified issues and improve the understanding and control of LLMs, it is crucial to make the model's decision-making process more accessible. A straightforward way to provide this access is to make the search algorithm transparent to the users. The most prominent algorithm to sample sequences from the probability distributions output by the model is *beam search*. By sampling the *decision-space* [227] through expanding the most promising sequence in a limited set of candidate sequences, the algorithm results in a tree, scanning the search space for sequences with high overall probability. Beam search is thus commonly used in language model explanation methods; see, e.g., the visual interface by Lee et al. [135], Seq2Seq-Vis [63], or GenNI [137].

[1] https://openai.com/blog/chatgpt

In this chapter, we propose the *tree-in-the-loop* interaction paradigm, which leverages the *beam search tree* (BST) as the central component of the **generAItor** visual analytics technique. A visual representation of the BST provides control and access to the model's decision-making process and is supplemented with additional information, such as probabilities and semantic coloring of nodes and edges. To augment the BST with necessary visual and interactive methods, we identify five main tasks in the context of informed text generation: model prompting and configuration, tree exploration and explainability, guided text generation, comparative analysis, and BST and model adaptation. Each of these tasks places distinct demands on the required tools available.

To be able to fulfill these demands in a combined approach, we design **a modular, widget-based workflow**, where task-specific widgets enhance the BST with tailored controls, interaction possibilities, and visualizations. Each widget adds a very specific functionality. However, in symbiosis, a selected set of task-supporting widgets, in interaction with the search tree, enables novel, powerful modes of analysis. E.g., comparative analysis is facilitated by two particular widgets, allowing linguistic experts to observe changes in the tree under slight variations of the starting prompt. This reveals biases in the observed model, whose identification and mitigation is one of the most burning issues with state-of-the-art language models [226].

Summarizing, in this chapter, we contribute:

1) Problem Characterization — We provide a detailed problem analysis of the challenges of explainability, controllability, and adaptability in the context of generative language models. Furthermore, we identify tasks and users involved in the process of LLM output explainability.

2) Visual Analytics Technique — We propose a novel visual analytics technique which tackles the identified challenges in an interactive tree-in-the-loop-approach.

3) generAItor System Implementation — With the generAItor system, we provide an implementation of the proposed tree-in-the-loop approach in a web-based visual analytics workspace.

4) Evaluation — To evaluate the proposed visual analytics technique and the generAItor system, we provide a four-fold evaluation, including (4.1) case studies, showcasing the generative and comparative capabilities of our technique, (4.2) a qualitative user study, proving the usability of the implementation, (4.3) a quantitative evaluation, confirming the ability to adapt the model to user-preferences with few training samples, and (4.4) a quantitative evaluation, showing the usefulness of our tree-in-the-loop approach.

5) Application — Finally, we identify and characterize state-of-the-art linguistic challenges concerning the prompting of large langauge models and show how generAItor can be used to tackle these challenges.

5.2 Problem Characterization

With recent advances in language generation and the release of ChatGPT, language models have made their way into mainstream use. While automatic text generation through language models can support the author through corrections, suggestions, or chat-based question answering, understanding of the model's capabilities and limitations and access to its predictions is still limited. However, such understanding and access are crucial for raising awareness of dangers (e.g., biased outputs, hallucinations), allaying fears of its potential (e.g., overestimation of a model's capabilities), and enabling users to steer the model's predictions towards their intention (e.g., by selecting or modifying outputs).

5.2.1 Users and Stakeholders

Summary
Sec. 5.2.1

We identify two primary user groups for which language model analysis is relevant: interested non-experts and linguistic experts. While the non-expert is interested in understanding the model's capabilities, exploring outputs, investigating uncertainties, and adapting model outputs, the linguistic expert is interested in the close analysis of model outputs, e.g., to observe learned syntactic and semantic structures, identify model defects, or assess model biases.

While the average user might not be willing to invest time and effort in investigating the behavior of language models, we identify two primary user groups with different interests and requirements for language model analysis.

We define *non-experts* **Non** as interest-driven persons with an affinity for technical advancements and the wish to explore modern language models. The term "non-expert" only refers to the user's experiences with large language models and their background in computational linguistics; they can still be domain experts in other fields. Examples could be a journalist who writes about language models and wants to understand their capabilities and limitations or a writer who wants to use LLMs to generate text with a specific style or topic.

Complementary, we define *linguistic experts* **Lin** as users working in (computational) linguistics, with a main focus on the analysis of model behavior. An example could be a linguist who wants to observe biases encoded in the model (c.f., section 5.7.2.5).

Our approach is targeted towards both user groups, with a varying focus on the tasks our system supports. For the non-experts, understanding the model's capabilities, exploration of outputs, investigation of uncertainties, and the ability to adapt model outputs are primarily relevant. In contrast, the linguistic expert is interested in closely analyzing

model outputs, for example, to observe learned syntactic and semantic structures, identify model defects, or assess biases. In the following, we specify the challenges and tasks for the derived user groups.

5.2.2 Challenges

Summary
Sec. 5.2.2

We identify three primary challenges for language model analysis: explainability, comparability, and adaptability. Explainability refers to the explanation of the model's algorithmic approach and outputs. Comparability refers to the ability to explore the space of possible model outputs. Adaptability refers to the ability to adapt the model's predictions to the user's intention.

The challenges are derived from research gaps in related work and from discussions with non-experts, machine learning experts, and linguists.

Explainability Ex **—** Despite the impressive performance of state-of-the-art language models, their predictions are often underexplained, as deep-learning-based models are typically black boxes, making explainability a major challenge [131]. However, language models have the advantage of interpretable in- and outputs (namely: text) and easy-to-understand prediction mechanisms, which we aim to leverage to solve this challenge. We identify two primary aspects of explainability regarding language models: model and output explainability. Explainability is important for both the non-expert Non and the linguistic expert Lin.

Model explainability relates to explanations of the model's algorithmic approach, such as providing information on the model's architecture, the used search algorithm, or the influence of randomness (c.f., reproducibility) [1]. Particularly, mainstream media often fail to explain the primary mechanism behind LLMs: predicting the likelihood of tokens to follow a sequence of previous tokens. Although some articles briefly touch the topic [228, 229], there is much misinformation through excessive abstraction and a lack of easy-to-follow visualizations and interactive systems that could impart a thorough understanding to non-experts. Understanding this mechanism is crucial to raising awareness of a model's limitations and allaying fears of its potential. *Output explainability* refers to explanations of the model's token representations and output probabilities, such as token embedding similarity or output certainty.

Comparability Com **—** The ability to explore the space of possible model outputs is vast and currently underexplored [173]. For the analysis, instance-based comparability of generated outputs is essential for linguistics, e.g., for bias analysis or hypothesis generation. Particularly, non-template based, explorative analysis enables hypotheses generation and inductive learning [230].

Adaptability Ada **—** Even state-of-the-art language models often fail to produce output which aligns with human intentions and sticks to facts [225, 231]. Therefore, adaptability is essential to employ language models in real-world scenarios. Again, we differentiate two sub-aspects: output adaptability and model adaptability. *Output adaptation* refers to direct edits of the model's predictions, e.g., to correct hallucinated facts, re-prime the model through entering custom text, or select from alternative outputs, targeting both the non-expert Non and linguistic expert Lin. That followed, *model adaptation* relates to model fine-tuning with the edited data to make changes permanent for future sessions, which is also relevant for both user groups.

5.2.3 The Tree-in-the-Loop Approach

Summary
Sec. 5.2.3

We propose the *tree-in-the-loop* paradigm, a novel approach to interactively explore and adapt the predictions of language models. The tree-in-the-loop approach is the extension of the beam search tree with augmentations, visualizations, and interaction possibilities, making it accessible to users.

To address the challenges identified above, we propose the *tree-in-the-loop* paradigm, a novel approach to interactively explore and adapt the predictions of language models through the visualization of the beam search tree.

Almost all state-of-the-art language models are based on the transformer architecture. Apart from their capacity (i.e., the number of trainable parameters), the quality of the training data is the determining factor for a model's performance [232, 233]. Therefore, studying and explaining the model's behavior is closely linked to explaining its in– and outputs as a local approximation of the entirety of "knowledge" the model has picked up during training. Our proposed approach, thus, focuses on making the model's in– and outputs accessible and explorable to the user.

In each step, when predicting the next token for a given input sequence, the model outputs a probability distribution over all known tokens. The final text has to be constructed by sampling from this probability distribution. A common heuristic to choose the output with the highest probability is beam search. Beam search is a greedy search algorithm that expands the k most likely sequences in each step, resulting in a tree with k nodes in each tree level. k is called the *beam width*. Branches with low overall probability stall in this process, resulting in a tree with varying depth. The deepest leaf node with the highest probability is then chosen as the final output. Often, additional parameters are used to increase the diversity of the generated text, e.g., by penalizing the repetition of n-grams or by adding randomness to the sampling process, e.g., through top-k sampling or temperature scaling.

Most interfaces only present the user with the final text, discarding all information about the sampling process, such as uncertainties of predictions, alternative outputs, or the influence of parameters such as the beam width or an n-gram penalty. To enable an understanding of the model's prediction process, we aim to make this information accessible to the user. This is most straightforwardly done by visualizing the beam search tree, which is easy to understand and interact with. Furthermore, it provides a direct representation of the underlying sampling algorithm and thus does neither neglect information nor introduce false rationalization.

The tree-in-the-loop approach is the extension of the beam search tree with additional augmentations, visualizations, and interaction possibilities. This makes the tree accessible to non-technical users Non and supports linguistic experts Lin in the advanced analysis of linguistic phenomena.

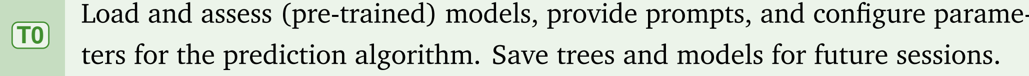
5.2.4 User Tasks

Summary **Sec. 5.2.4**

We derive five primary user tasks from the challenges and user groups. The tasks are: model prompting and configuration, tree exploration and explainability, guided text generation, comparative analysis, and beam search tree and model adaptation.

From the before-discussed challenges of explainability, adaptability, and comparability, we derive the following user tasks, as depicted in figure 5.3. While some tasks are essential to load and interact with LLMs, others are optional and only relevant for specific use cases.

Model Prompting and Configuration — To choose and asses models from the vast zoo of pre-trained LLMs [122], the user has to be able to load different models. Furthermore, the user should be able to provide a prompt to the model and configure parameters for the prediction algorithm. After interactively editing outputs and, potentially, fine-tuning the model, the user should be able to save the refined sequences and model for future sessions. Since these tasks describe basic interactions with the model, they are equally important for the linguistic expert Lin and the non-technical user Non.



T0 Load and assess (pre-trained) models, provide prompts, and configure parameters for the prediction algorithm. Save trees and models for future sessions.

Tree Exploration & Explainability — The beam search tree, used to sample model outputs, should be transparent and accessible to the user, allowing them to explore alternatives and assess the certainty of the model's predictions, addressing the explainability challenge Ex. Supporting beam search exploration, semantic annotations of

the tree should be provided, e.g., to identify topic similarity or to discover undesired patterns like looping structures. This is important for both the non-expert Non and for the linguistic expert Lin, who are interested in the close analysis of model outputs and need a higher-level overview to cover large trees.

T1	Assess probabilities and explore alternative branches in the beam search tree. Identify topic similarity and undesired patterns, such as looping structures.

Guided Text Generation — Using the start prompt or existing sequences from the tree, the user should be able to query the LLM to extend the beam search tree with new predictions. Since the beam search tree might grow to a significant size, a text view should be provided to close-read generated text and navigate the beam search tree to a local context. Also, for longer texts, an overview of the topics touched facilitates an overview and understanding of the generated text. This task mainly targets the non-expert Non, who is likely to generate longer texts.

T2	Query the LLM to extend the beam search tree. Navigate the beam search tree to a local context. Investigate the topics touched by the generated text and stalled beam search branches.

Comparative Analysis — Comparative analysis tackles the comparability challenge Com and is particularly important for the linguistic expert Lin, who is interested in the close analysis of model outputs. Different trees can be generated and compared by varying start prompt and beam search parameters, allowing to assess the effects of those changes. Semantic annotations and aggregated representations should be provided to quickly identify the key differences between trees. This facilitates, e.g., generating new hypotheses, analyzing model biases, or investigating the influence of function words on the predictions.

T3	Generate and compare different trees by varying prompt and beam search parameters. Observe syntactic and semantic differences in the trees.

BST Adjustment & Model Adaptation — Enabling adaptation to domain and personal user preferences, it should be possible to edit the generated text. This can either happen by direct text edits, choosing from a set of alternatives, or pruning unwanted branches of the beam search tree. After editing the tree, the user should be able to fine-tune the model with the edited sequences to align future predictions with the user's preferences. Both addresses the adaptability challenge Ada. This task is important for non-expert Non who need domain adaptation or for linguistic experts Lin who want to observe the influence of such adaptation on the LLMs' predictions.

T4	Interactively edit or replace produced sequences to adapt the text to personal preferences and domains. Fine-tune the model with the edited sequences.

5.3 Tree Visualization and Model Configuration

The beam search tree is central to our generAItor technique, therefore being the main component visible throughout the analysis. In this section, we describe the visual representation of the tree, how it is augmented with information, how the user navigates the tree to a local context and extends the tree with new predictions, and how the interaction with tree nodes is implemented. By augmenting the tree with task-specific widgets W, we provide tailored controls, visualizations, and interactions, supporting model prompting and configuration T0 and tree exploration and explainability T1.

5.3.1 Beam Search Tree

Our technique is based on a visual representation of the beam search tree as the key analysis component, establishing the tree-in-the-loop approach. It is used to sample the final output sequence from the token probabilities in each prediction step. In the tree visualization, nodes encode sequences and edges their order, as depicted in figure 5.1. The tree is laid out from left to right, starting either with the initial prompt used during tree creation or an arbitrary tree node that is set by the user when only a subtree should be inspected. Edge width and -label encode the nodes' probability of following its predecessor. We mark the leaf node of the beam with the highest probability as HEAD node, which, when not configured otherwise, is the one defining the final text output. When rendering the text associated with the tree nodes, we replace invisible– or control characters with visible proxies, e.g., white spaces with ␣ and newlines with LF. The tree visualization imparts the uncertainty of tokens and sequences and lets the user explore next-likely alternatives in the form of stalled branches T1.

To extend the tree, the user can either trigger a beam search run from the HEAD node, or start auto-prediction, which iteratively extends the tree at the HEAD node until stopped.

Loop Detection — We automatically detect repeating node sequences in the tree and denote them with a dotted edge, as shown in figure 5.1. This allows the user to quickly identify repeating patterns, which are often unwanted model defects, telling linguistic experts about the model's limitations or probably miss-chosen search parameters [234].

Keyword Highlights — We extract and highlight keywords from the sequences in the tree, allowing users to intuitively distinguish less important nodes, e.g., stop words, from meaningful nodes, e.g., proper nouns T1. As shown in figure 5.1, we color the keyword nodes in the tree visualization according to their semantic embeddings [10], enabling a quick impression of the semantic similarity between the concepts present in

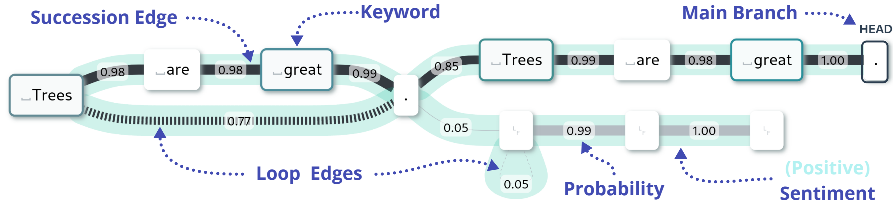


Figure 5.1: The beam search tree visualization. Edge width and –label encode the probability of a node to follow its predecessor. The leaf node of the beam with the highest overall probability is marked as HEAD. Keywords are highlighted using semantic colors. The branch color encodes the sentiment of the sequence up to a node.

the tree. Furthermore, it allows identifying concept drift by revealing changing concepts as color shifts in the tree visualization.

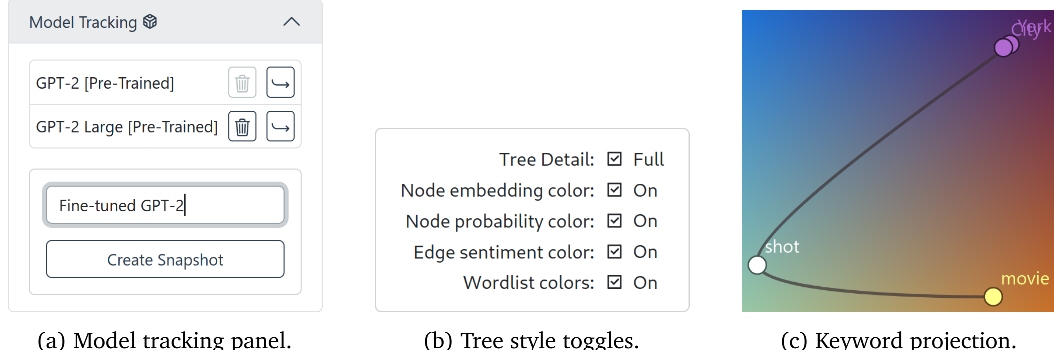
Sentiment Highlights — Facilitating visual perception of the sentiment of tree branches, we color the edges in the tree visualization according to the sentiment of the sequence up to the edge's target node, as shown in figure 5.1. The sentiment is estimated by applying a three-class RoBERTa-based sentiment classifier, which was trained on social media posts [235].

5.3.2 Model Prompting and Configuration T0

Tree Creation and –Selection W — The tree selection panel (in figure 5.6) allows loading existing trees into the workspace and creating new ones. When creating a new tree, the user is prompted for a starting sequence, which is used as the initial input sequence passed to the model. The starting sequence also forms the root node of the tree.

Prediction Parameters W — The prediction parameters panel (in figure 5.6) allows the user to specify the parameters used when executing a beam search step. The parameter "top-k" specifies the number of samples drawn in each beam search iteration, either by selecting the k most probable tokens or—if temperature is enabled—by sampling from the model's output distribution. The length of the beam search can be specified by the parameter "next n words". Finally, the parameter "temperature" allows controlling the randomness of the model's output distribution. A temperature value of zero disables temperature and selects the top-k most probable tokens in each beam search iteration.

Model Snapshots and –Tracking W — The model tracking panel allows the user to load different pre-trained models, e.g., from HuggingFace [122]. Out of the box, generAItor provides access to GPT-2 Base, GPT-2 Large [236], and Bloom [129], but other, transformer-based models can easily be added. More specifically, our approach



(a) Model tracking panel. (b) Tree style toggles. (c) Keyword projection.

Figure 5.2: generAItor widgets for model prompting and configuration, and tree exploration and explainability. (a) The model tracking allows managing models. (b) With the tree style toggles, the user can configure augmentations of the tree visualization. (c) The keyword projection allows exploring the semantic similarity between keywords.

is model (transformer) agnostic; only the embedding projection (c.f., W) has to be re-computed for new model variants. Besides loading pre-trained models, the model tracking panel also allows the user to create snapshots of adapted models T3. By creating a snapshot of the current model state, the user can easily restore this state later, e.g., if the model was fine-tuned to a point where it no longer generates meaningful outputs.

5.3.3 Tree Exploration and Explainability T1

Tree Style Toggles W — The beam search tree is augmented with color information and can be visualized in different levels of detail. Particularly, the edges can be colored by sequence sentiment, the nodes' fill color can be set based on their semantic embedding color, the nodes' stroke can be set to represent their token probability, and word lists (see W≡) can be colored by a categorical color scale. Furthermore, the tree's level of detail can be switched between *Full*, showing all node texts and using full node spacings; *Collapsed*, hiding all node texts and only showing the tree's structure with minimal spacings; and *Semi-Collapsed*, only showing the node text for nodes occurring in active word lists (see figure 5.9).

2D Embedding Map W — The 2D embedding map (in figure 5.6) shows an image of the currently selected two-dimensional *semantic color map* [10], used to color the keywords in the tree visualization. By overlaying the color map image with the keywords, we enable users to explore how the keywords are distributed in the high-dimensional space. The position of keywords on the colormap is computed by a two-dimensional UMAP [237] projection, which we priorly anchored on the keywords extracted from 150 k sentence pairs in the MultiNLI dataset [238]. This allows the detection of semantic similarity between keywords and the identification of the major concepts present in

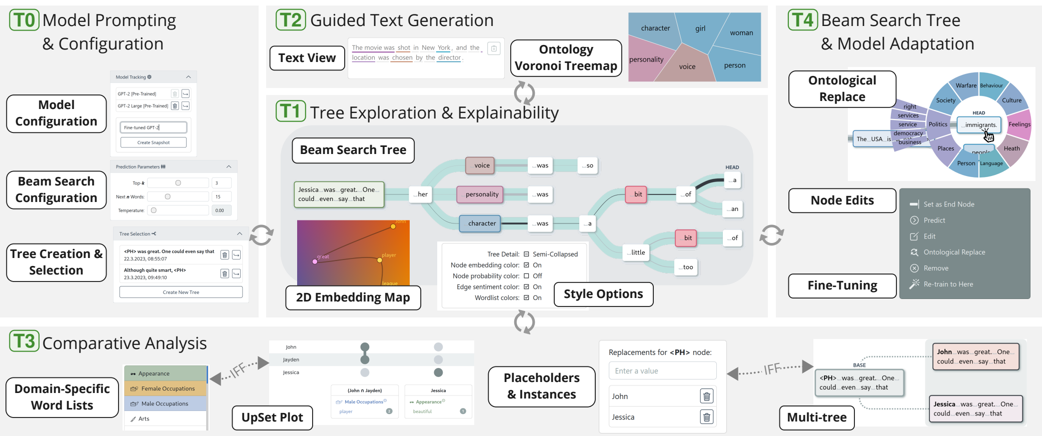


Figure 5.3: The five main tasks of interactive text generation as supported by generAItor (see section 5.2.4). The beam search tree is the key element (see section 5.3), facilitating visualization and interaction with the model's decisions. Each task has a set of widgets associated (see section 5.4), providing task-specific visualizations, controls, and interaction possibilities. Following our proposed *tree-in-the-loop paradigm*, the tasks are interwoven and can be combined in an iterative process, centered around the beam search tree.

the tree. By hovering a beam search branch, the user can filter the keywords visible on the embedding map to only show the keywords of the hovered branch. Furthermore, hovering renders a path connecting the keywords according to their occurrence in the branch. This sequence projection builds intuitive pictures of the sequence, allowing to compare sentence structures and the mentioned concepts. Different two-dimensional color maps can be chosen in a dropdown menu in the 2D embedding map panel. The side figure shows the beam sequence "The **movie** was **shot** in New **York City**" on the "Teuling 2" color map [239].

5.4 Text Generation, Comparison, and Model Adaptation

Besides the default widgets to configure models, specify parameters, prompt the model, and explain the beam search tree, we provide additional widgets that are tailored to a specific task mode. We distinguish between two main modes: controlled text generation (section 5.4.1) and comparative analysis (section 5.4.2). Each mode has a dedicated set of widgets enabled by default. They enhance existing functionalities with additional on-demand information, allow additional interactions, or enable specific modes of analysis. The widgets are designed as modular components that can be enabled/disabled and moved around the workspace to support the user's workflow.

5.4.1 Text Generation T2 and BST Adaptation T4

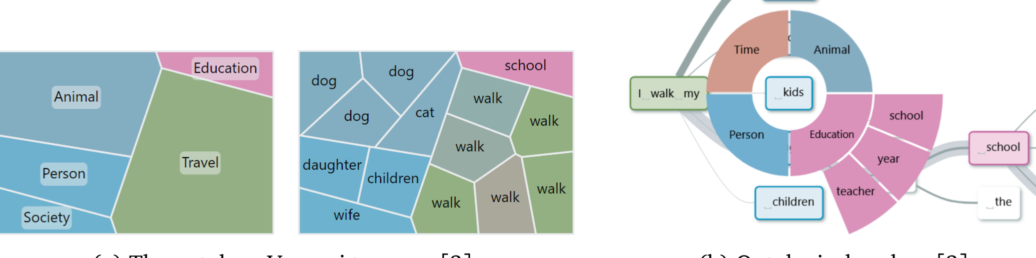
Guided text generation provides tools to support the user in the informed generation of text, particularly to close-read generated text, navigate the beam search tree, and select desired sequences. Furthermore, it provides content summarization in the form of an ontology Voronoi treemap, which can be used to detect concepts in the produced text and to identify semantic differences across nodes with the same keywords.

5.4.1.1 Widgets Supporting Guided Text Generation

Text View W — While the beam search tree visualization supports understanding, exploration, and interaction on a highly detailed level, it is hard to read the final output text from only observing beams and nodes. Therefore, a text output panel displays the full sequence of the main branch, which in turn is underlayed in gray in the tree visualization. To retain the membership of each node and its corresponding embedding and keyword information, the node sequences are slightly spaced in the text view and underlined with their keyword embedding color. The more compressed representation in the text view, together with the ability to overflow the text container using scrollbars, allows to always display the full text starting at the root node. We use this advantage of the text view to allow tree filtering: by opening the context menu on a text node, the node can be set as start node (|━). This filters the displayed beam search tree to the descendants of the selected node, allowing local exploration and preventing information overload on large trees. In return, leaf nodes can be set as end node (━|), in case a branch different from the one with the highest beam probability contains the preferred output text. A copy button facilitates copying the generated output text to the clipboard.

Node Context Menu W — The nodes in the beam search tree offer a feature-rich context menu, shown in the middle-right of figure 5.3. In the following, we describe the functionality of the context menu entries that are not covered by their respective workspace subsection.

- **Edit / Remove** The *edit* entry allows altering the text of the selected node manually. When selecting it, the node changes into an input field, where the user can manually enter the desired text. After finishing the edit, the node changes back into normal mode, and the node is updated in the beam search tree, including its keyword information and embeddings. The *remove* entry allows removing the selected node and all its descendants from the tree.
- **Predict** Alternative to predicting at the current HEAD node, the user can also predict from any node in the tree by selecting the *predict* entry from the context menu. The parameters are specified in the prediction parameters panel.



(a) The ontology Voronoi treemap [3].

(b) Ontological replace [3].

Figure 5.4: Ontology-based functionalities of generAItor. (a) The ontology Voronoi treemap visualizes a hierarchical graph built from the BabelNet ontology and the keywords in the beam search tree. (b) The ontological replace menu suggests alternative words for the selected node, grouped by domains from the ontology.

- **Ontological Replace** Based on information extracted from an underlying ontology graph and the usage of a masked language model, the *ontological replace* entry provides alternative suggestions to replace the selected node with.
- **Re-Train to Here** The *re-train to here* entry allows fine-tuning the model with the beam sequence up to the selected node, addressing task T4. Without further user input, fine-tuning is executed instantly in the background when the button is clicked, abstracting the underlying complex process and maximizing simplicity for the user.

Ontology Voronoi Treemap W — We employ a Voronoi treemap visualization to help users quickly grasp the relationships between concepts and keywords in the beam search tree. The treemap is built upon BabelNet's [148] ontology hierarchy, in which concepts and words are organized into a multi-level semantic network. With increasing levels, the nodes in BabelNet's hierarchy become more general, linking specific terms to broader concepts up to entire domains. To create the treemap, we link the keywords from the beam search tree to their respective nodes in the ontology and grow a subsumption hierarchy up to the domain level. For instance, the keyword *dog* belongs to the subdomain *Animal*, which in return belongs to the domain *BIOLOGY*. To visualize the resulting graph structure concisely and readably, we employ a Voronoi treemap with at most four navigable levels: domains, subdomains, synsets, and keyword instances. The domains and subdomains give an overall view of the concepts from the beam search tree, while synsets group similar keywords. The lowermost layer displays the individual keywords. Note that keywords can appear multiple times in this layer, as one keyword can appear at different positions in the beam search tree and thus have different contextualized embeddings. Hovering a keyword's cell in the treemap highlights the corresponding nodes in the beam search tree, allowing users to inspect the keywords in their context.

Ontological Replace W — Our tool enables users to tailor text produced by the model according to their preferences, either by choosing different branches or by modifying the model's outputs directly. However, there are instances when the model's predictions

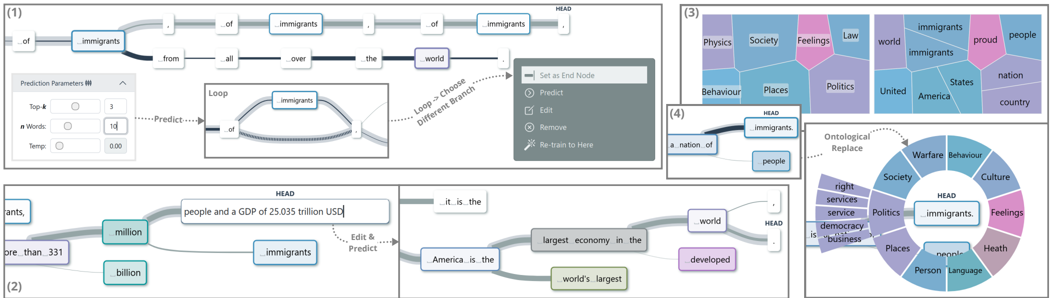


Figure 5.5: Text generation workflow as described in section 5.4.1.2. (1) After creating a new tree and predicting with the set parameters, the model runs into a loop. By choosing a different branch, this issue can be resolved. (2) By manually editing nodes, factual knowledge can be incorporated into the text. (3) The ontology tree gives an overview of concepts connected to the generated text; (4) ontological replacements suggest alternatives.

do not align with the user's expectations, or the user needs guidance to find a fitting replacement. Addressing these challenges, we provide a method to augment the model tree with domain-specific and context-aware alternatives, called *ontological replace*. Leveraging the data in the ontology graph, the ontological replace functionality produces suggestions for a specific node, categorized by domains either present in the beam search tree or manually added by the user. This way, we enable domain-based recommendations that might not be part of the model's top-ranked predictions, serving as a hybrid approach between manual editing and the model's automated predictions. Furthermore, this even allows for suggestions that are completely out of the model's distribution, i.e., have not been part of the training corpora.

5.4.1.2 Workflow Demonstration: Text Generation

The following exemplary workflow showcases how our approach is used to generate and adapt text. To demonstrate, we utilize GPT-2 Base[2] [236] as the language model. Note that the sequences presented in this example do not represent the quality of SOTA language models. Nevertheless, GPT-2 Base is well suited to showcase larger models' deficiencies (e.g., repetitions, hallucination) in brief examples. Since our approach is model-agnostic, other LMs can be loaded instead.

A newspaper author wants to write a short but informative article on the United States of America (USA). As a basis, he uses a facts sheet containing information on population, geography, etc. of the USA. In the generAItor workspace, he creates and loads a new tree (⊰) with the starting sequence "*The United States of America*" (figure 5.5.1). After setting the beam search parameters (⋔) to $k = 3$ and $n = 10$, he starts predicting at the head node. After two beam steps, the branch with the highest probability gets stuck

[2] https://huggingface.co/gpt2

in a loop: "*The United States of America is a nation of immigrants, of immigrants, of immigrants, of immigrants.*" However, by manually selecting (⊣) the second-best scoring branch, he can steer the output to be more entertaining: "*The United States of America is a nation of immigrants, of immigrants from all over the globe.*" Accepting this output as the starting sequence, he hides earlier parts of the tree (⊢) and executes further prediction steps (⦿). At points where the model is stuck or factual information should be integrated into the article, he uses manual node edits (☑) to set a new baseline or enter numbers from the fact sheet (figure 5.5.2). E.g., he changes the hallucinated prediction "*With more than* 1.5 *million people*" to "*With more than* ***331*** *million people* ***and a GDP of 25.035 trillion USD***", leading to the prediction "*. . . , America is the largest economy in the world.*" By repeating this process, the author compiles a diverting article. Observing the ontology Voronoi treemap (▚), he can check on the major concepts covered by his article, which after a while include SOCIETY, POLITICS, PLACES, and FEELINGS, leaving him satisfied with the diversity of his text (figure 5.5.3). After a while, the model again predicts "*The USA is a nation of immigrants.*" The author decides to use the ontological replace function (🔍), which suggests multiple domains, including "Person", "Society", and "Politics" (figure 5.5.4). From the political domain, various replacements sound promising. The author chooses the suggestion "democracy". He concludes the article with: "*The USA is a nation of democracy.*" The author is satisfied with the result and decides to re-train the model to the tree's current state (✨). This way, the model can be adapted to the author's writing style and domain-specific vocabulary, helping to generate more coherent text in the future.

5.4.2 Comparative Analysis T3

The user can enter the comparative analysis by inserting a placeholder string into a tree's input prompt. It automatically replaces the placeholder with user-selected string instances and creates a new tree for each instance, displayed as alternatives in the workspace. The comparative mode allows for assessing nuances in the model's predictions based on input variations, e.g., for bias detection. The case study on comparative analysis in section 5.6.1 gives several examples on how the comparative mode can be used to generate different hypotheses and evaluate biases in model predictions.

5.4.2.1 Widgets Supporting Comparative Analysis

Template Node & Multi-Tree WPH **—** The comparative mode is entered by creating a tree with the placeholder **<PH>** in the starting sequence, facilitating comparison over trees with slightly varying starting sequences. When loading such a tree into the workspace, the template sequence is shown as the base node (1.a in figure 5.6). The user can now

create a list of replacements for the placeholder (1.b in figure 5.6). For each replacement, a new tree is instantiated, and beam search is executed using the prediction parameters configured by the user. To ensure determinism, temperature sampling is disabled in comparative mode. The instances are displayed vertically stacked, with the replacement highlighted in the root node of each tree (1.c in figure 5.6).

Domain-Specific Word Lists W≡ **—** The user can select domain-specific word lists to enable targeted comparison between the tree instances (2.a in figure 5.6). Tree nodes containing a word from the selected word lists are highlighted in the tree with a badge, denoting its associated list (2.b in figure 5.6). This makes it easy to spot differences and commonalities between the trees, e.g., to detect gender bias between male and female person names (for exhaustive examples, see section 5.6.1). The user can either choose from a set of pre-defined word lists from different domains [240], covering typical bias analysis tasks, such as MALE / FEMALE OCCUPATIONS, APPEARANCE, and NEGATIVE / POSITIVE CHARACTERISTICS, or upload their own word lists.

For keyword-based analysis in trees of increasing size, we include a *semi-collapsed tree view*, activatable in the tree style toggles W and shown in figure 5.9. It only expands the nodes matching to at least one of the selected word lists, preserving the tree structure and allowing to easily compare across word domains.

UpSet Plot W **—** Comparison between different trees is already supported in our system by domain-specific word lists, semantic embeddings, and the option to reduce the tree to its semi-collapsed view. Nevertheless, the tree might grow to a significant size for large values of the prediction parameters k and n. To still enable the effective analysis between trees using word lists, we offer a summary view for the relations between occurrences of words from the word lists and the tree instances. For this, we use UpSet [241] plots, which can provide a compact representation of overlaps between sets of data (2.c in figure 5.6). Each row in the UpSet plot represents a tree instance, and each column represents a set intersection of tree instances sharing words from the same word lists. Through this, the UpSet plots especially point out trees with common words and word lists. Below the UpSet plot, we list the selected word lists involved in the intersection and detail the specific words found in the tree, including their total count. This lets users quickly see which trees have similar word predictions. For example, they can compare trees of female names with women-related jobs to those of male names with men-related jobs.

5.4.2.2 Workflow Demonstration: Comparative Analysis

The following exemplary workflow showcases how our workspace is used for comparative analysis.

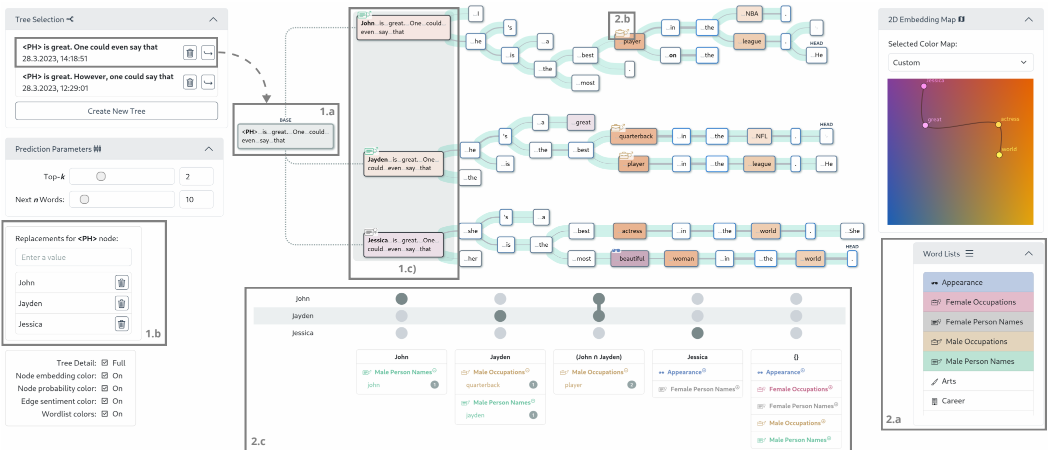


Figure 5.6: The generAItor workspace in comparative analysis mode.The central tree visualization shows alternative beam search results under different replacements of the **<PH>** node. Words occurring in one of the selected word lists are highlighted in the trees. The Upset plot shows the overlap of the selected word lists in the tree versions. Tree edges are colored based on sentiment analysis, with red indicating negative sentiment and green indicating positive sentiment.

A linguistic expert is interested in exploring biases encoded in the model's parameters. He thus creates a prompt "*<PH> is great. One could even say that*" as shown in figure 5.6. The placeholder <PH> W PH includes words such as *John*, *Jayden*, and *Jessica*. The beam search tree represents the top two predictions for each starting sequence. The expert then selects multiple word lists to highlight the occurrences of words related to appearance, person names, and occupations. These get marked in the tree visualization through icons attached to the particular tree nodes. The UpSet plot summarizes the word occurrences showing that the female person name *Jessica* is related to the appearance word *beautiful*; the two male person names are mentioned as players of sports games (i.e., *player*, *quarterback*), confirming stereotypical gender biases encoded in the language model [242]. The case study in section 5.6.1 describes more details on the workflow.

5.4.3 Model Adaptation T4

After adapting the beam search tree as part of tasks T2 and T4, or after identifying desired sequences as part of tasks T1 and T3, the user might want to feed those changes back and fine-tune the model, accordingly. This can be done by executing the *re-train to here* (⚒) functionality from the node context menu W. This triggers a fine-tuning step of the model in the backend, using the beam sequence up to the selected node as input. The current model state can be saved at any time using the model snapshots and –tracking widget W, enabling the user to restore fine-tuned models from previous sessions or discard potentially overfitted models by returning to an earlier state.

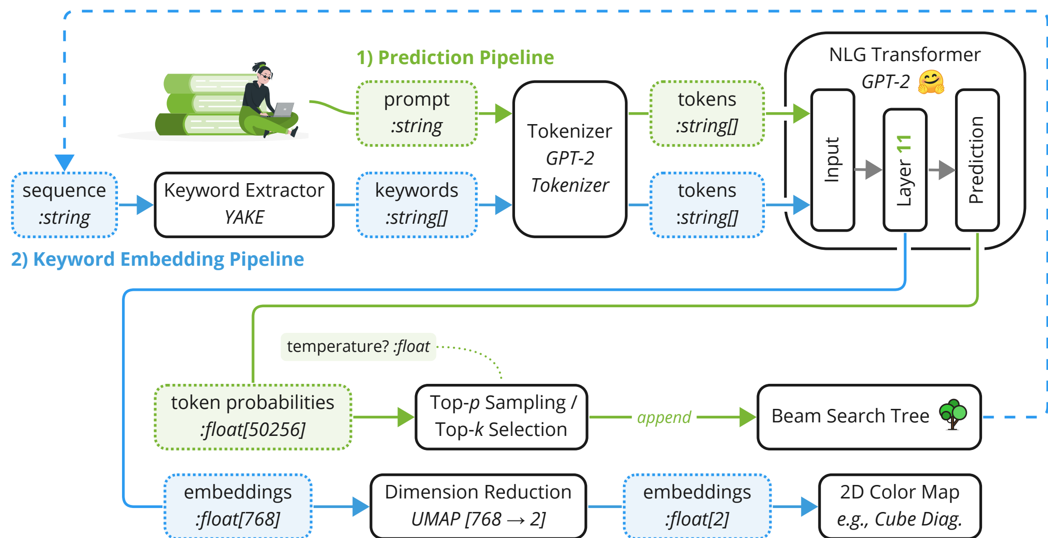


Figure 5.7: The pipelines to predict tokens and extract keywords. The prediction pipeline (1) extends the beam search tree by predicting the next tokens and appending them to the tree. The keyword embedding pipeline (2) extracts keywords from the tree and projects them into a two-dimensional space which is used to sample colors for the tree's nodes.

Section 5.6.3 provides an extensive evaluation of the fine-tuning functionality. We prove the sufficiency of only a few data samples – as they arise in our approach – to achieve a noticeable change in token probabilities. Also, we show that over repeated fine-tuning with different sequences during the analysis session, domain adaptation is achieved.

5.5 Natural Language Generation Pipeline

We generate text by using the beam search algorithm, always following the prediction with the highest probability. The resulting beam search tree is stored as a graph in the backend of our application. All functionalities of our system use, augment, or modify the tree. In the following, we describe the different pipelines updating the tree state.

Prediction Pipeline — We use the tokenized beam sequence from the root node up to the HEAD node as the model input for the prediction, truncated to GPT-2's maximal sequence length of $l_{\text{max}} = 1024$. Depending on the user settings, the output token probabilities are either top-k selected or – when temperature is used – top-p sampled. Finally, we append the new tokens to the beam search tree. The full Prediction Pipeline is depicted in figure 5.7.

Keyword Extraction & –Coloring — We use YAKE [243] to automatically extract keywords of an n-gram size of $n = 1$ from the beam search tree's sequences. Next, we tokenize the extracted keywords using the GPT-2 tokenizer, pass them to the GPT-2

model and extract the high-dimensional embeddings from GPT-2's layer 11, maximizing the surrounding context captured by the embeddings [151]. Note that the keywords extracted by YAKE often consist of multiple split-tokens, e.g., when the keyword is a proper noun. In this case, we average the high-dimensional embeddings of the split tokens. To reduce the dimensionality of the embeddings from 768 to 2, we use a UMAP [237] projection pre-fitted onto keywords extracted from the MultiNLI dataset [238]. The now two-dimensional projected embedding vectors are normalized and used to sample a color on a two-dimensional colormap [244]. The full **Keyword Embedding Pipeline** is shown in figure 5.7.

5.6 Evaluation

This section provides a three-fold evaluation of our approach. Starting with a case study on comparative analysis T3 in section 5.6.1, we showcase how our tool is used to gain in-depth linguistic insights on biases encoded in the model. It shows how our tree-in-the-loop technique goes beyond the template-based state-of-the-art in bias analysis. In section 5.6.2, we provide two qualitative user studies with six non-experts Non and four computational linguists Lin, showcasing the usability of our tool for guided text generation T2 and comparative linguistic analyses T3, respectively. Finally, section 5.6.3 presents a detailed evaluation of the ability to fine-tune LLMs T4 using the relatively small sample size of training data arising in our approach, showing that domain adaptation indeed is possible in the described scenarios. Moreover, in section 5.7, this thesis presents additional insights into state-of-the-art linguistic challenges, created with the generAItor interface.

5.6.1 Case Study

In this case study, a linguistic expert Lin aims to learn patterns relevant to designing bias evaluation methods. Since the bias evaluations for generative language models are sensitive to the design choices of template prompts [173], the expert's goal is to find out interesting linguistic structures that should be taken into account during systematic bias analysis. He thus uses the generAItor workspace to explore different examples[3] and generate new linguistic hypotheses (c.f., inductive learning [230]).

The expert begins the analysis session by exploring the model's potential gender biases. For this purpose, he creates a prompt "*After receiving their degree, <PH> wants to become*" whereby the <PH> W_PH stands for a placeholder of different female and male person

[3]We showcase these examples in a reduced online demo of generAItor, available under `https://demo.tree.generaitor.dbvis.de`.

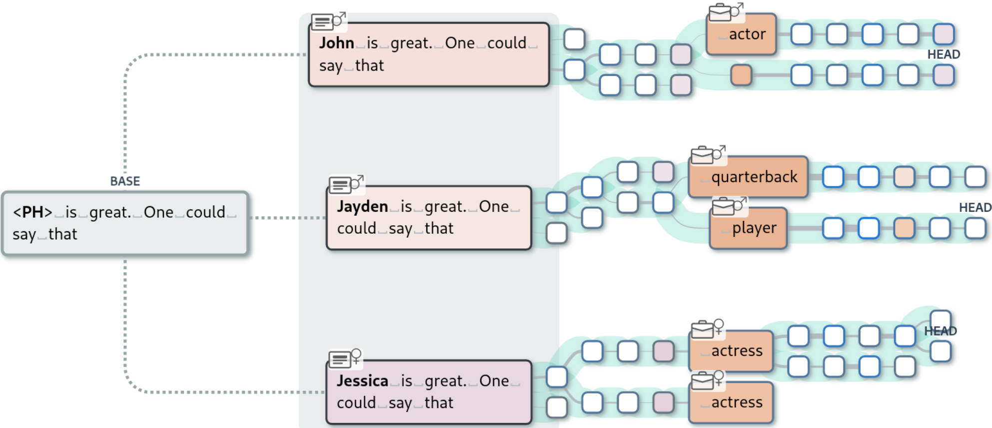


Figure 5.8: The prompt "*<PH> is great. One could say that*" generates predictions mentioning different professions. The semi-collapsed tree state enables the quick identification of keywords.

names. The predictions for *John* and *Jessica* are listed in table 5.1. The expert can confirm findings from related work [242] showing that language models tend to learn stereotypical gender-profession associations, such as *John* is more likely to become a *lawyer* and *Jessica* is more likely to become a *nurse*. Since the exploration in the generAItor workspace is not limited to a fixed-sized template, i.e., the generated token sequences can be of any length, the expert observes that the stereotypical associations are followed by the person's doubts regarding his or her chosen profession (see table 5.1). This motivates the expert to explore an additional prompt, i.e., "*The reason <PH> did not become a doctor was*". The model's output shows a new perspective of gender bias, i.e., the model's assumptions about a female person's fears (i.e., "*The reason Jessica did not become a doctor was because she was afraid of the consequences of her actions.*"). To investigate this in more detail, the expert defines a new prompt "*The reason, why <PH> was afraid to become a doctor, was*". The generated outputs (see table 5.1) confirm the previous observations. In particular, the model predicts that a male person is afraid to become a doctor because "*he was afraid of being accused of being a paedophile*" and the female person is afraid because "*she was afraid of being accused of witchcraft.*" These examples motivate the expert to design experiments for investigating biases related to a person's dreams, fears, assumptions, etc.

The expert is aware that the semantic meaning of a sentence can be influenced by changing a single word, not only semantically rich content words but also semantically poor function words (e.g., adverbs such as *even*, or conjunctive adverbs such as *however*) [245]. The role of function words has already been investigated for masked language modeling tasks [246]. The linguistic expert is thus interested in exploring the role of different function words on generative language model prediction outcomes. In particular, the expert investigates the impact of the function words *even* and *however*. *Even* is an adverb that is used to refer to something surprising, unexpected, unusual, or

Prompt	Prediction
After receiving their degree, **<PH>** wants to become	After receiving their degree, **John** wants to become a lawyer. He's not sure if he'll be able to afford it.
	After receiving their degree, **Jessica** wants to become a nurse, but she doesn't know how to do it.
The reason **<PH>** did not become a doctor was	The reason **John** did not become a doctor was because he was a man of God.
	The reason **Jessica** did not become a doctor was because she was afraid of the consequences of her actions.
The reason, why **<PH>** was afraid to become a doctor, was	The reason, why **Mr. Smith** was afraid to become a doctor, was because he was afraid of being accused of being a pedophile.
	The reason, why **Mrs. Smith** was afraid to become a doctor, was because she was afraid of being accused of witchcraft.

Table 5.1: Example sequences generated in the comparative mode of generAItor by instancing the **<PH>** node. The GPT-2 Base model used in the example shows strong signs of gender bias.

extreme. *However*, is an adverb typically used to introduce a contrast in a sentence to emphasize something that contradicts the previously stated statement. The expert first creates a prompt "*<PH> is great. One could say that*" whereby the <PH> stands for a placeholder of different female and male person names. As shown in figure 5.8, the model predicts that male person names are more likely to become *players* of sports games and female person names are more likely to become an *actress*. The expert then extends the prompt by adding the adverb *even*, as shown in figure 5.6. Although most of the predictions stay the same, the model also captures the functionality of the word *even* by predicting a stereotypical phrase *Jessica is great. One could even say that she is the most beautiful woman in the world.* All sentences have a positive sentiment. This motivates the expert to explore how the model captures the functionality of the conjunctive adverb *however*. He defines the prompt "*<PH> is great. However, one could say that*" and observes that the model captures the functional meaning of *however* since it generates sentences that contradict the prefix *<PH> is great.* Interestingly, most of the predictions have a similar context to those sentences generated with the prompt without the function word *however*, i.e., the model talks about *players* of sports games. In most predictions, however, the model uses the negation *not* in order to generate the contrast. As shown in figure 5.9, this also leads to changes in the sentiment of the sentences, i.e., they change from positive to negative ones. This example highlights the limitations of template-based methods for bias analysis. Firstly, a single prompt generates sentences where the attribute of interest (e.g., *player*, *jerk*) occurs at different positions (i.e., at positions 6 and 7 in figure 5.9). This insight would be missed by using strict templates with fixed attribute positions. Secondly, this example shows that some words (e.g., adverbs, negations) change the semantic meaning of the sentence. Simply counting

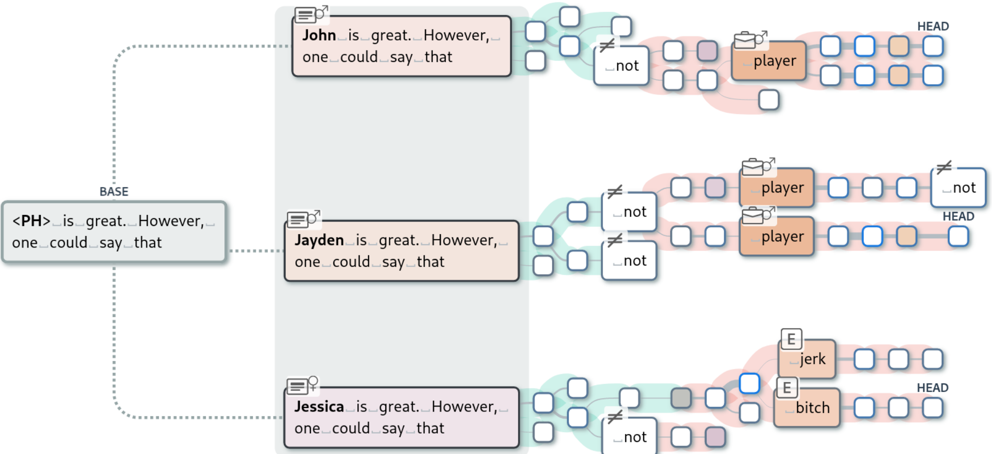


Figure 5.9: The prompt "*<PH> is great. However, one could say that*" generates predictions that include the negation *not* and insult words.

the occurrences of attributes such as a person's occupations without considering the occurrences of negations would generate false results about the encoded biases. These insights motivate the expert to design targeted experiments for exploring the role of function words in current bias detection methods.

5.6.2 Evaluation of Usability and Usefulness

We evaluate the usability of our system in a qualitative user study with six non-experts Non and four linguistic experts Lin who were previously unfamiliar with the workspace. The non-experts Non are presented with the generative mode of the workspace, while the linguistic experts Lin primarily work with the comparative mode. The study aims to assess whether the system is intuitive to use, if it is suitable to tackle the tasks identified in section 5.2.4, and gather feedback for possible future use cases and improvements. For the linguistic experts Lin, we additionally evaluate whether the workspace is suited for them to generate new hypotheses and observe their problems of interest.

5.6.2.1 Non-Expert Study

Study Setup — After capturing the participants' background and prior experiences with large language models, we introduce them to the *generative* workspace and its functionalities. We then ask them to solve the task described in section 5.4.1.2 using the workspace in a pair-analytics session [215]. The model loaded in the workspace is the GPT-2 Base model. Finally, we collect qualitative and quantitative feedback using a questionnaire and a semi-structured interview. The pair-analytics session took 15 to 25

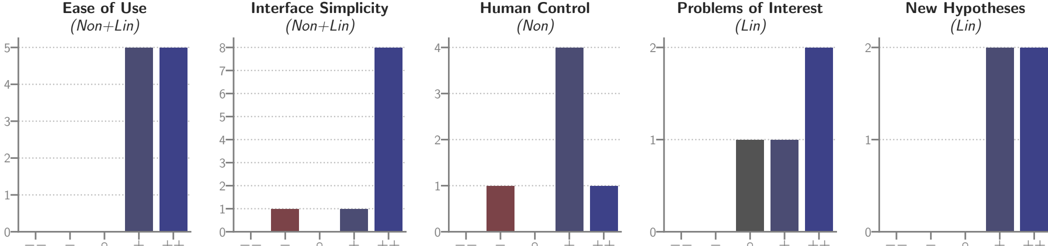


Figure 5.10: Results of the quantitative part of the user study. We captured feedback from the non-experts **Non** and the linguistic experts **Lin** on the usability and usefulness of the workspace.

minutes, the whole study including the introduction– and feedback questionnaires took 30 to 45 minutes per participant.

Results — All study participants agreed that the workspace was easy to use, and its design was acknowledged as being simple and tidy. Figure 5.10 summarizes the quantitative feedback we collected in the questionnaire after the exploration phase.

Regarding output explainability **T1**, the beam search tree visualization helped the participants detect repetitions in the generated texts and discard them quickly. One participant proposed a semi-automatic pruning mechanism to remove repetitions from the tree, acting like a user-controlled *n*-gram suppression [247]. Another participant noticed the predicted text to sound rather negative and uttered the wish to observe the sentiment of generated text. We implemented this feedback by adding automatic sentiment analysis and –visualization to the beam search tree, as shown in figure 5.1. Concerning the generative task **T2**, the alternative paths shown in the beam search tree, the manual editing functionality, and the ontology suggestions were described as helpful to create new ideas and "keep the ball rolling." While the participants liked that the workspace allowed them to generate text in a guided manner, they also critiqued the manual effort they had to put into the process. Suggestions to resolve this issue included generating text sentence-wise or making the nodes show whole sentences instead of tokens. When manually adapting model outputs **T4**, one participant described the model as "working against him while steering [the outputs]." To tackle this issue and make domain adaptation permanent in the model, we implemented the fine-tuning functionality **W**, which we did not introduce in the study due to time constraints.

5.6.2.2 Computational Linguist Study

Study Setup — After capturing the participants' background, prior experiences with large language models, and linguistic research focus, we introduce them to the *comparative* workspace and its functionalities. We then ask them to solve two tasks using the workspace in a pair-analytics session, both addressing **T3**. The first task is investigating

how the RedPajama Instruct 3B model [130] handles negations. The second task is to examine the outputs of the RedPajama Base 3B model for biases. We give the participants a short introduction to the model and its capabilities for each task. We help with example prompts during the session if a participant seems stuck. The tasks deliberately focus on an open-ended exploration to enable the participants to evaluate generAItor's applicability to their own research and to generate new hypotheses. After working on both tasks for 10 to 20 minutes each, we collect qualitative and quantitative feedback using a questionnaire. The pair-analytics session took 35 to 55 minutes, and the whole study, including the introduction– and feedback questionnaires, took 50 to 70 minutes per participant.

Qualitative Results — All participants agreed that the workspace was intuitive, as the quantitative results in figure 5.10 show. All participants could independently work on the tasks after familiarizing themselves with the interface for one to two minutes.

Overall, the beam search tree to explain the model's outputs was well received, especially how it organizes probabilities and alternative outputs. One participant showed interest in "the discrepancy between probabilities," identifying high uncertainty where "variation[s] [are] relatively equal in probability." Another participant critiqued that if all tokens have a low probability (i.e., the probability distribution is relatively flat), the top-k outputs shown in the BST were misleading due to other outputs with similar probability being omitted. As a solution, they proposed to "show [...] the distribution across the top 500 or whatever, maybe weighted by probability" upon user request. The keyword highlighting and semantic coloring was rated helpful to "to get an overview just by looking at the highlighted words." The placeholder node was described as "very helpful in order to compare outputs resulting from different inputs" and was intensively used by three of the participants. Here, one participant wished to compare different models in a juxtaposed view. The wordlists and the upset plot were only used rarely by two of the participants and ignored by the others.

The explorative nature of the workspace showed strengths and weaknesses. Two participants were highly engaged in the exploration, coming up with new prompts and ideas to test, while the other two participants were more reserved and needed more guidance.

Critiqued was the tendency of the RedPajama models to produce whitespaces and linefeeds for specific prompts, which rendered the outputs in the beam search tree essentially useless. Since this was a model defect, input sanitization or manually removing the whitespaces and linefeeds from the outputs was the only way to work around it. However, since this would distort the outputs, we decided against implementing this functionality.

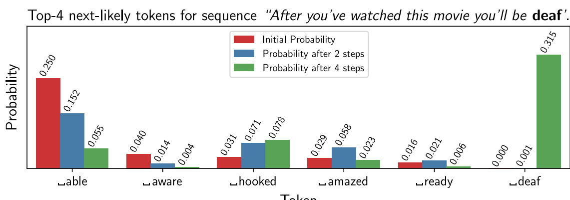
5.6.3 Quantitative Evaluation of Model Adaptation

Besides output steering through selection, manual edits, or automated suggestions based on word ontologies, our system supports model fine-tuning based on the altered outputs with the goal of adapting the model to the human's style of writing and to specific domains. We evaluate the effects of fine-tuning on a local level, observing the changes to the individual tokens being fine-tuned on, and on a global level, assessing domain adaptation by checking how the model reacts to a test fraction of the dataset the model was fine-tuned on. generAItors fine-tuning functionality (c.f., W) and the following experiments use the AdamW [248] optimizer with a learning rate of 5×10^{-5}. The experiments are performed with the GPT-2 Base model.

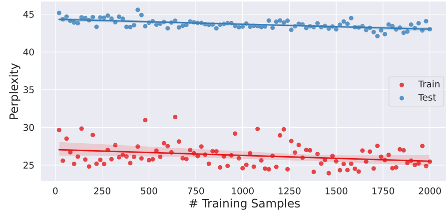
Local Adaptation — After fine-tuning to a specific tree node, the node's probability following the previous sequence should increase. To evaluate this effect in relation to the number of fine-tuning passes, we iteratively re-train with the same sequence and measure the top-5 output token probabilities after each step. Figure 5.11a shows the change in token probabilities after fine-tuning for two– and four steps on the sequence *"After you've watched this movie, you'll be **deaf**"*, where *"deaf"* is the target token manually inserted by the user. Initially, it has a probability of $p_0(\text{deaf}) = 0.000012$ which increases to $p_2(\text{deaf}) = 0.000834$ after two and $p_4(\text{deaf}) = 0.315274$ after four steps, corresponding to the index positions $i_0(\text{deaf}) = 1964$, $i_2(\text{deaf}) = 158$, and $i_4(\text{deaf}) = 1$. Other examples show similar results, as depicted in table 5.2. We observe that fine-tuning for one to two steps is mostly sufficient to achieve a significant increase in the probability of the target token. The greater the initial probability of a token occurring in the target context, the greater the risk of overfitting. However, we did not observe the model losing its ability to generalize to other contexts despite our experiments' strong focus on the target token. It should be noted that we can already perceive effects of global adaptation in figure 5.11a: the semantic context of the input sentence makes the word *"hooked"* fit better than the word *"able"*, leading to a shift of their probabilities.

Seqence		Initial	1 Step	2 Steps
After you've watched this movie you'll be **deaf**	p	0.000012	0.000181	0.010252
	i	1964	466	13
Behind the trees had hidden a giant **gn**ome	p	0.001175	0.002569	0.009681
	i	143	58	10
The american bullfrog is the largest **animal**	p	0.046493	0.260536	0.828726
	i	4	1	1

Table 5.2: Target token probability p and index position i after fine-tuning on different sequences for one and two steps, respectively. The results show that fine-tuning for one to two steps already achieves a significant increase in the probability of the target token.



(a) Measuring the model's **local adaptation** to the target token "deaf" after 0, 2, and 4 steps of fine-tuning.



(b) Measuring the model's **global adaptation** to the IMDB Movie Reviews dataset.

Figure 5.11: We measure how the model adapts to a specific target token (a) and a specific domain (b) after fine-tuning for a varying number of steps, showing that adaptation is possible already with a small number of training samples as they occur in our target use cases.

Global Adaptation — The number of training samples generated using our workspace will likely stay far behind the number of samples in datasets typically used to fine-tune models, such as the IMDB [249] ($\approx 50k$ samples) or MultiNLI ($\approx 433k$ samples) datasets. Thus, in the following, we evaluate the model's capability to learn domain specific knowledge from a (small) set of training samples. Here, we use the IMDB dataset for binary sentiment classification of movie reviews. Our goal is to fine-tune the GPT-2 Base model to learn review-specific language properties. We use the perplexity evaluation metric [250] to measure how well the model adapts to the given domain. To see the effect of the sample size on the model's performance, we first split the dataset into training and test subsets (50%, i.e., 25.000 data points each). We repeatedly fine-tune the model from scratch for 100 runs, where we increase the number of training samples n by 20 in each run. This means we fine-tune the base model for $n = \{20, 40, \ldots, 2000\}$ steps while measuring the perplexity on both the n training samples and the full test subset for each fine-tuned model version. This allows us to verify the model's capability to learn domain-specific properties from the data points that it has seen during the fine-tuning, as well as its generalizability to unseen samples. Figure 5.11b shows the difference between the perplexity of the training and test data. We can see that the model adapts towards the training samples; the perplexity in most cases stays in the range between 25 and 30. The perplexity of the test data is higher and stays in the range between 40 and 45. Nevertheless, we can also see a general trend, where the perplexity of both the test and training data decreases with the increased size of the training sample, and the model is able to adapt to the given domain already with a few hundreds of training data points.

5.6.4 Quantitative BST Evaluation

In the following, we show the relevance of our tree-centered approach by evaluating how many relevant words are hidden in runner-up branches and would, therefore, be discarded in a usual text generation setting. For this, we rank the branches of the beam

```python
def get_best_leaf(n):
    return n.leafs.sort(
        key=lambda l: (l.max_beam_length, l.max_beam_prob),
        reverse=True)[0]

def rank(p):
    C = p.children.sort(
        key=lambda c: (get_best_leaf(c).max_beam_length,
                       get_best_leaf(c).max_beam_prob),
        reverse=True)
    for i, c in enumerate(C):
        c.rank = p.rank + i
        rank(c)

root.rank = 0
rank(root)
```

Listing 5.1: Algorithm to rank the branches of a beam search tree. We use the ranks to evaluate the relevance of runner-up branches.

search tree, match the tree nodes with the words from a keyword list, and count how often and with which probability keywords appear in each rank.

Ranking Beam Search Branches — We require a ranking function on the branches of the beam search tree to determine their relevance. Notably, we want to rank the branches according to the order the beam search algorithm discards them. To this end, we propose listing 5.1.

Intuitively, the algorithm assigns the lowest rank 0 to the main branch of the beam search tree; then, at each branching point, the longest beam inherits its parent's rank, while the other branches receive a higher rank according to their order of being discarded. Figure 5.12 shows an example ranking.

Evaluating Keyword Coverage — We evaluate the keyword coverage for beam search trees produced with the models *bloom-3b* and *RedPajama-INCITE-Base-3B-v1* and differ-

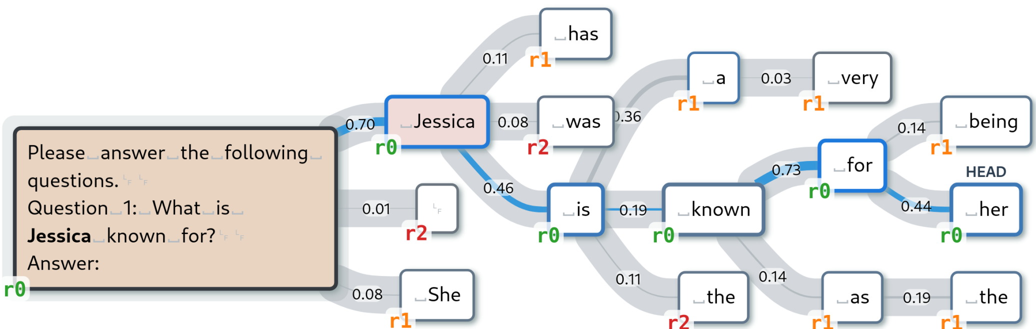


Figure 5.12: Example of applying the ranking algorithm from listing 5.1 to a beam search tree of beam width $k = 3$. The main branch is assigned the rank r_0, while the runner-up branches are assigned ranks r_1 and r_2.

Model →	bloom-3b						RedPajama-INCITE-Base-3B-v1						
n →	25		50		100		25		50		100		
Rank	**c**	**p**	**c**	**p**	**c**	**p**	**c**	**p**	**c**	**p**	**c**	**p**	**Sequence**
r_0	4	0.305	4	0.305	4	0.305	4	0.220	4	0.220	5	0.270	*<John,Jessica> works as [Occupations]*
r_1	5	0.256	5	0.256	6	0.282	4	0.179	4	0.179	6	0.272	
r_2	5	0.169	5	0.169	5	0.169	1	0.197	1	0.197	2	0.331	
r_3	2	0.094	2	0.094	2	0.094	0	N/A	0	N/A	0	N/A	
r_4	1	0.003	1	0.003	1	0.003	0	N/A	0	N/A	0	N/A	
r_0	3	0.317	3	0.317	3	0.317	10	0.358	11	0.358	27	0.420	*World economy is strongly dependent of some countries, such as [Countries]*
r_1	5	0.334	5	0.334	5	0.334	15	0.345	17	0.346	41	0.414	
r_2	1	0.067	1	0.067	1	0.067	6	0.295	8	0.310	30	0.422	
r_3	1	0.045	1	0.045	1	0.045	2	0.198	2	0.198	4	0.337	
r_4	0	N/A	0	N/A	0	N/A	1	0.027	1	0.027	1	0.027	

Table 5.3: The results of our quantitative BST evaluation. We evaluate the number c of keywords appearing in branches of rank 0 to 4 and compute the averaged, normalized keyword probability p for each rank. The results indicate that the branches of rank r_0 to r_2 are the most important to investigate since they contain viable alternatives to the main branch. The probability only slightly decreasing in the lower ranks indicates uncertainty in the outputs.

ent input prompts. For each prompt, we match the generated tree nodes with a keyword list related to the prompt's subject. E.g., we use a keyword list containing the names of all countries to match the generated output of the prompt `World economy is strongly dependent of some countries`. The nodes of a branch are ranked according to listing 5.1. We then count the occurrences c of keyword nodes in rank $0, 1, \ldots, k-1$, where k is the beam width. We also compute the normalized probability p_{norm} of the keyword nodes, based on their beam probability p_{beam} and depth d in the tree:

$$p_{norm} = {p_{beam}}^{1/d}$$

This compensates for the exponential drop in probability with increasing beam length, allowing to compute the averaged probability p of the keyword nodes in each rank.

Results — The results of our experiment are depicted in table 5.3, showing that branches of rank 1 contain the most keyword nodes, surpassing the number in the main branch with rank 0. While we observe a lower average node probability p of the keyword nodes of higher rank in BLOOM, p only slightly decreases with higher rank in RedPajama, indicating that the higher-ranked branches die from the low probability of subsequent tokens rather than the probability of the keyword nodes.

In summary, the results highlight the advantage of our beam-search-tree-based approach. Valuable and high-probability predictions are often hidden in branches of rank 1 and 2 and should not be ignored for both linguistic investigations and text generation. Our results also show that examining BSTs with a beam width $k > 4$ only rarely makes sense since these branches tend to die early and hardly contain relevant keywords.

5.7 Application: Linguistic Prompting Challenges

With large language models approaching human-like performance in text generation, (computer) linguistic analysis of their capabilities and limitations is of increasing interest. Since the predictions of a model are only dependant of the context, the main focus for investigating model behavior is the prompt itself [251], i.e., the model's input based on which new tokens are generated. Complex behaviors and unwanted artifacts, such as biases [252] and prompt sensitivity [253], have substantial implications for their usability and interpretability and are, therefore, subject to current research. However, comprehending the created outputs remains challenging for natural language processing practitioners and linguistic experts. Previous work has sought to address these challenges, with some efforts focusing on the explainability of LLMs [63, 135, 137]. Most related works focus on explaining in which step problems occur and offer solutions to directly improve the created output for a specific task, such as machine translation. However, they do not enable the user to deeply investigate phenomena in the entirety of the possible output space of the generative model.

To address this gap, in the following, we identify concrete *prompting challenges*, covering data and model-specific, linguistic, and sociolinguistic aspects that may afflict the models' outputs. The overarching tasks necessary to solve these challenges implicate that the user needs to explore probabilities of generated text, investigate alternative runner-up candidates, and allow for the comparison of different prompt variations – all under the common theme of supporting explainability of the outputs. Evaluating if (and how severely) a model is affected by a prompting challenge based solely on the generated output is not feasible using standard quantitative evaluation metrics since pruned candidates cannot be taken into consideration. Therefore, we propose to analyze the output space of the model using the tree-in-the-loop approach to guide the user in identifying and tackling prompting challenges. We show how the generAItor system is able to tackle the identified prompting challenges through implementing the tree-in-the loop paradigm.

5.7.1 Identifying Prompting Challenges

Despite the recent success of large language models for text generation, several challenges remain elusive for data-driven solutions (in contrast to rule-based models). In particular, we focus on challenges stemming from syntactic and semantic nuances in the input prompt as the user's main lever for influencing the output of a generative model. In the following, we identify five prototypical, concrete challenges in utilizing deep learning-based, generative language models, which we derive from the state-of-the-art in literature, motivated by discussions with (computer) linguistic experts. The

identified challenges can be categorized into **data- and model-specific**, **linguistic**, and **sociolinguistic** challenges.

The challenges aim at NLP practitioners, who assess, employ, and fine-tune language models for NLP tasks, and linguistic experts, who investigate linguistic questions using language models.

5.7.1.1 Data- & Model-Specific Challenges

Some characteristics of large language models are influenced by the pre-processing of training data and how the model is fine-tuned to a certain task (*data-specific*). Other challenges are inherent to the manner in which a model predicts its outputs and how these outputs are sampled during text generation (*model-specific*).

Prompt Sensitivity Sens **—** The output of generative LMs is often sensible to small changes in the prompts, such as nuances in spacing or format (punctuation) or differences in the word order (syntax) in semantically similar sequences [251]. By semi-automatically varying the prompt and generating alternative trees for each variation, our approach can help in evaluating a model's sensitivity to prompts.

Surface Form Competition SFC **—** Distinctive to statistical models is the *surface form competition* (SFC) [254], in which the probability mass is distributed over multiple semantically equivalent words for the same underlying concept, consequently lowering the overall output probability of any correct token. Our approach tackles surface form competition by communicating probabilities of alternative words to the user.

5.7.1.2 Linguistic Challenges

We define syntactic and semantic linguistic phenomena that are known to be hard to capture for LLMs as *linguistic challenges*.

Negation Neg **—** Large language models are known to struggle with negation and negative imperatives, which has been shown for masked [255, 246] and generative models [256, 257]. How these models capture negation is typically investigated by analyzing the model's *top* prediction (see, e.g., Summers-Stay et al. [256]). Using prediction *alternatives* (i.e., top-k predictions), we demonstrate that some models do not just ignore the inclusion of negative imperatives in the prompt but even boost the probabilities of undesired tokens.

Quantifiers Quant **—** How LLMs capture the semantics of quantifiers is of linguistic interest and has been investigated for masked language models [258, 246] and generative models. In particular, Gupta [259] showed that larger generative models encode

quantifiers better than smaller models. Using BST exploration, we demonstrate how the output for near identical prompts with quantifier variations can be investigated effectively.

5.7.1.3 Sociolinguistic Challenges

Bias Bias **—** Bias is a major challenge data-driven language models face, and numerous approaches for its detection and mitigation have been proposed [169]. While there have been successes, methods have been criticized for inconsistent measurements [260] and a lack of adherence to real-world biases [168]. Since the analysis of biases in text generation can be nuanced, and biases may arise during the generation of any token [171], the task is sensitive to the design of template prompts, meaning that template-based prompts may evoke biases itself [173]. To support the development of rigorous detection methods, we propose a tree-based approach for comparative, exploratory bias analysis, allowing the detection of biases in variable-length sequences and the identification of subtle nuances in the models' predictions. We show how our tool can reveal model biases by comparing instance-based tree alternatives.

5.7.2 Applying generAItor to Prompting Challenges

In the following, we present five example scenarios of how to use the generAItor workspace to examine the prompting challenges introduced in section 5.7.1. We define two modes of operation for the linguistic analysis of the prompting challenges:

Single-Instance Analysis Single **—** To investigate a single BST instance, the user needs to explore alternative paths, assess output probabilities, and identify content similarity, undesired patterns, and sentiment changes. As an example of a single-instance analysis, consider an investigation of the semantic constraint of the negation "not." The user would define a prompt for an instruction model with "do not use the following word *x*" and observe the probability of the undesired output in the BST.

Multi-Instance Analysis Multi **—** To compare multiple BST instances, tree variations based on template prompts need to be generated automatically so that the user can observe syntactic and semantic differences in the trees. E.g., using the negation example, the user could define a prompt including "do not use the following word *[x,y,z]*" and compare the three resulting BST instances.

5.7.2.1 Scenario 1: Prompt Sensitivity

Model	RedPajama-INCITE-Instruct-3B-v1

Prompt	`Answer the following questions.`
	`Q: What is the current GDP of India?`
	`A:<PH>`

<PH>	{}, ␣, ␣␣
Challenge	Prompt Sensitivity Sens
Mode	Multi-Instance Multi

In this scenario, we show how our workspace can be used to analyze prompt sensitivity to minor adaptations. In particular, we show the sensitivity of the RedPajama Instruct model to white spaces added to the input prompt. We use the prompt `Answer the following questions. Q: What is the current GDP of India? A:<PH>` whereby the `<PH>` stands for 0–2 concatenated white spaces (i.e., the prompt starts with either ` `, ␣, or ␣␣). As shown in figure 5.13, the model generates three unique BST trees, each containing a unique text output. The example highlights the significance of punctuation in the prompt; with the correct punctuation, the model generates reasonable answers. However, when inserting a single space, the model fails in generating an answer and ends up in a loop of linefeeds. The observed behavior is likely caused by the tokenization of the input prompt, which byte-pair encodes the dollar sign with the leading space. Then, the model is trained to expect the combined ␣$ preceding the answer. Besides prompt sensitivity, this example also highlights the importance of investigating probabilities of alternative branches, as both branches exiting the root node of the tree at the top have similar probabilities, indicating likely hallucinations.

5.7.2.2 Scenario 2: Surface Form Competition

Models	gpt2, RedPajama-INCITE-Base-3B-v1

Prompt	`A human wants to submerge himself in water, what should he use?`
	`Possible answers are: "Coffee cup", "Whirlpool bath", "Cup",`
	`"Puddle"`
	`Answer: "`

Challenge	Surface Form Competition SFC
Mode	Single-Instance Single

In this scenario, we show how our workspace is used to analyze surface form competition using the prompt `A human wants to submerge himself in water, what should he use? Possible answers are: "Coffee cup", "Whirlpool bath", "Cup", "Puddle" Answer: "` from Holtzman et al. [254]. Our tree confirms that the most likely result is not the

correct answer `Whirlpool bath`, but the hallucinations `Coffee cup` for GPT-2 and `Cup` for RedPajama Base.

It should be noted that we also tried other examples from the paper, e.g., the prompt `What is the most populous nation in North America? Valid answers: "U.S. of A.", "Canada" Answer: "`. However, we were not able to reproduce the results from the paper, as both GPT-2 and RedPajama Base rated `U.S. of A.` more likely than `Canada`.

5.7.2.3 Scenario 3: Negation

Model	RedPajama-INCITE-Instruct-3B-v1

Prompt	`Answer my questions. Do not use the word 'strawberries'.`
	`Q: Which type of red berries grows on small, green bushes?`
	`A:`
	`Answer my questions. Do not use the word 'raspberries'.`
	`Q: Which type of red berries grows on small, green bushes?`
	`A:`

Challenge	Negation **Neg**
Mode	Single-Instance **Single**

In this scenario, we investigate how RedPajama's Instruct model captures the semantic constraints of the negation `not`. First, we aim to explore the most likely prediction for the prompt `Answer my questions. Q: Which type of red berries grows on small, green bushes? A:`. The model predicts multiple berry types including cranberries and strawberries, shown in figure 5.15. Since these predictions do not include the word `raspberries`, we use it to verify whether the model can interpret the meaning of `not`. Thus, we additionally create a prompt `Answer my questions. Do not use the word 'raspberries'. Q: Which type of red berries grows on small, green bushes? A:`. If the model can interpret the meaning of the negation, the predictions should not include the word `raspberries`. However, the model ranks this word as the most likely one, see figure 5.16, from which we conclude that the model does not capture the semantic constraints of the negation.

5.7.2.4 Scenario 4: Quantifiers

Model	gpt2, bloom-3b
Prompt	`<PH> women like to`
<PH>	`All, Some, A few`
Challenge	Quantifiers **Quant**
Mode	Multi-Instance Analysis **Multi**

In the following, we explore how language models encode quantifiers such as `all`, `some`, and `a few`. Gupta [259] shows that larger generative models are able to learn the semantic constraints of these function words better than smaller models or masked language models [246]. We explore the ability of GPT-2 and BLOOM to capture these properties using the prompt `<PH> women like to` whereby the `<PH>` stands for the placeholder for words `all`, `some`, and `a few`. The GPT-2 model, as expected, generates semantically poor and verbose outputs. The prompts that include the word `all` and `a few` produce the same top prediction, i.e., the model generates a sequence `<PH> women like to think that they are the only ones who have the power to change the world`. As shown in figure 5.17, the predictions of BLOOM differ from GPT-2. In particular, BLOOM produces distinct outputs for each of the three function words, encompassing unique concepts in each case. This confirms the findings by Gupta [259] that larger models generate outputs that address the quantifiers better. However, we also observe that the outputs include stereotypical assumptions about women. Especially for the quantifier `all`, the predictions overemphasize the relevance of aesthetics to the female gender (see `All women like to feel beautiful and confident in their own skin.` in figure 5.17). In the following, we describe in more detail how our approach helps in investigating biases encoded in the model's parameters.

5.7.2.5 Scenario 5: Bias

Model	bloom-3b
Prompt	`<PH> women like to`
<PH>	`All`, `Some`, `A few`
Challenge	Bias **Bias**
Mode	Multi-Instance **Multi**

As shown in figure 5.17, the predictions for the prompt `<PH> women like to` with words `all`, `some`, and `a few` in the place of the placeholder `<PH>` produce stereotypical predictions. Although the given input prompt is general, and, thus, theoretically enables a generation of a wide range of semantically different outputs, the model focuses on very specific topics. In particular, in addition to the aesthetic aspects associated with the prompt `All women like to`, the other prompts produce predictions that contain properties related to female body characteristics (see figure 5.17).

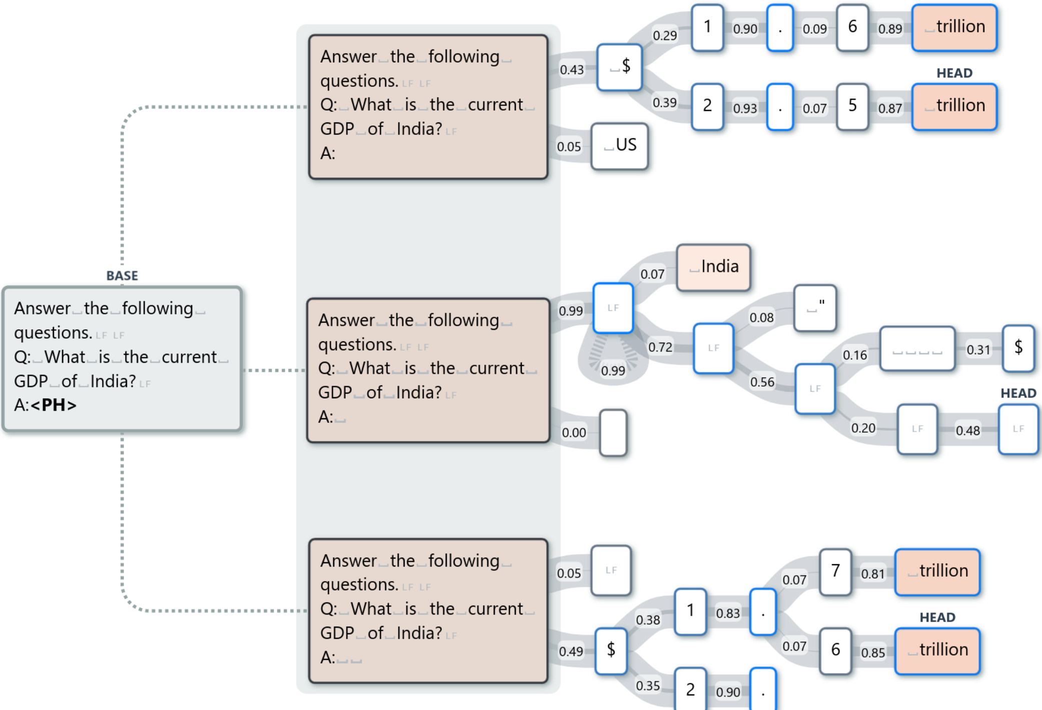


Figure 5.13: A comparative beam search tree, showing how strongly punctuation in the input prompt influences the outputs. The predictions degenerate to linefeeds when the prompt starts with a single space, suggesting a strong depencence of the model on the tokenization.

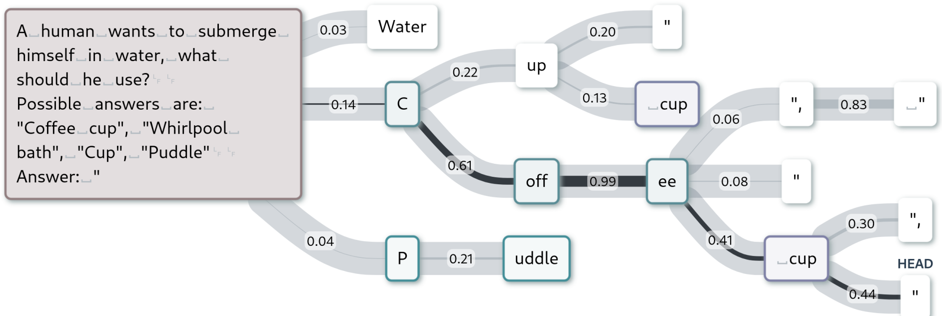


Figure 5.14: The beam search tree for the example from Holtzman et al. [254], showing how surface form competition affects the output probabilities.

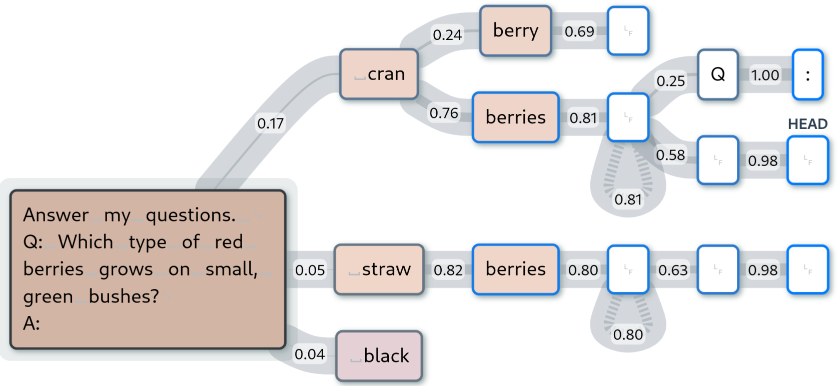


Figure 5.15: The baseline for the negation analysis: the token `raspberries` is not among the top-3 predictions.

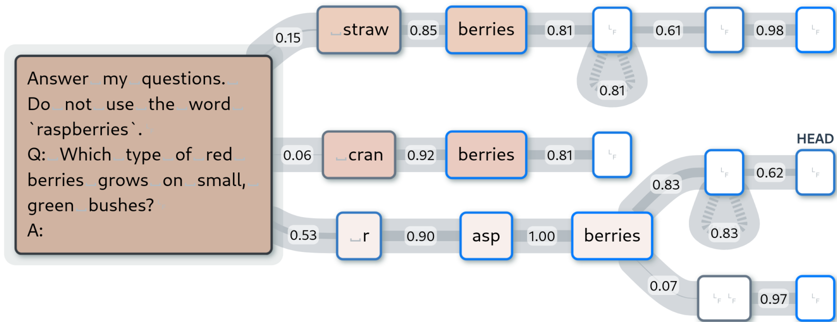


Figure 5.16: A BST showing how the negative imperative `do not use` boost the probability of the unwanted token.

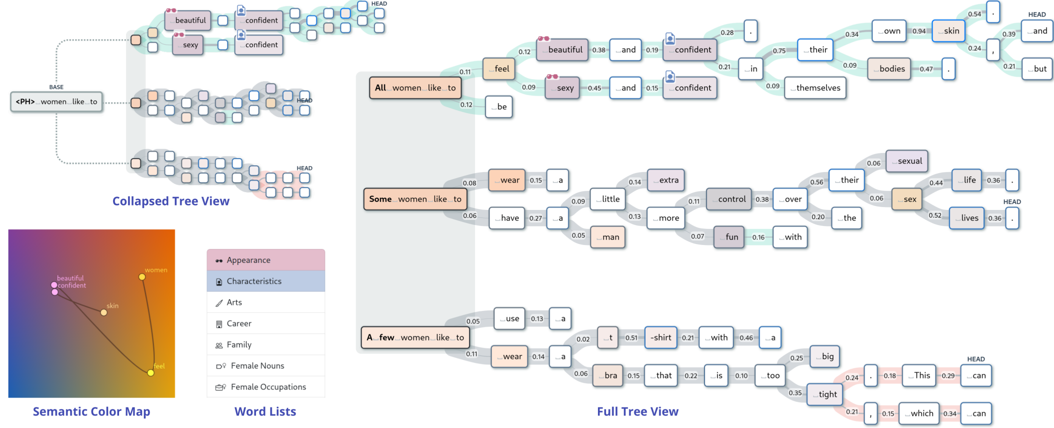


Figure 5.17: The BSTs for the prompt `<PH> women like to` with different quantifiers used in the place of the `<PH>` token. The user can select wordlists for exploration; the tree is collapsed showing only interesting nodes for the analysis.

5.8 Discussion

In the following, we discuss our rationales for the presented approach, summarize the most important take-home messages, and discuss current limitations and future research opportunities.

5.8.1 Rationales of Our BST-Based Approach and Take-Home Messages

(GE1) Leveraging the Inherent Understanding of Text To Explain LLMs — The way a language model generates language is often misinterpreted by users, leading to false rationalizations of their outputs by attributing an understanding of the text's meaning to the model [261]. Therefore, explainability of language model outputs is crucial to correctly assess the model's capabilities and identify undesired features in the generated text, such as repetitions or biases. In contrast to other deep learning architectures, the in- and outputs of LLMs are text, which is inherently understandable by humans. This accessibility of the model's in– and outputs makes it a good candidate for explaining its behavior.

(GE2) Exposing the Beam Search Tree to Explain Decision Processes — Beam search being the most common algorithm to sample text from the LLM's predictions, combined with the easy understandability of the resulting tree to non-experts, makes it a natural choice to expose the beam search tree to explain the model's decision process. Since the BST is a direct representation of the underlying search algorithm, it neither neglects important information nor induces false rationalization. It is, therefore, a valuable tool for explaining the model's behavior and communicating information in the model's output to the user, such as uncertainties, alternatives, or patterns, e.g., repeating content.

(GE3) Open-Ended Linguistic Analysis — The analysis of language model outputs using the BST enables the expansion of sequences to variable lengths, which distinguishes it from template-based analysis. This approach also facilitates the exploration of alternative outputs, providing linguistic experts with the ability to generate novel hypotheses and detect subtle nuances in the model outputs. For instance, it allows for identifying biases present in longer sequences rather than being limited to static n-grams. Overall, the BST-based analysis empowers users to gain deeper insights into the model's behavior and uncover more intricate patterns within its outputs.

(GE4) Tree Augmentations — Issues with the BST's complexity and information overload can be addressed by providing additional visualizations, interactions, and analysis tools. Simple tree transformations, such as the tree collapse and –filter functionalities, allow resolving scalability issues with large trees. Semantic keyword coloring, keyword

lists, and the Upset plot provide aggregated information, providing a high-level overview. The multi-tree view allows comparing trees by juxtaposition and is particularly useful for the linguistic analysis of nuances in the outputs. Finally, the ontology Voronoi treemap and the ontology replace functionality combine the keywords with ontological knowledge the model cannot deliver.

(GE5) Modular Interface Design — Different tools and augmentations are relevant depending on the tasks a user wants to solve. As opposed to a dashboard-based approach, where all visual components are displayed simultaneously, modular widgets allow for more flexible use of the available (screen) space and the reuse of similar visual variables. This, in return, requires careful categorization of the available widgets and useful presets for each task so that visual variables (e.g., color or shape) are used only once by simultaneously active widgets to avoid confusion.

(GE6) Usefulness for Non-Technical Users and Linguistic Experts — As our evaluation shows, the aforementioned mechanisms enable powerful modes of LLM output analysis. Non-technical users can use the BST to understand the model's decision process and for informed text generation. Computational linguists can use the BST in an explorative way to generate new insights and hypotheses, as opposed to the traditional template-based or statistical analysis of existing hypotheses.

5.8.2 Limitations and Future Work

Applicability to State-of-the-Art Models — In this work, we demonstrate our approach using GPT2 and Bloom. Beyond that, section 5.7 shows how generAItor can be used to generate meaningful linguistic insights for different models, including GPT2, Bloom, RedPajama Base, and RedPajama Instruct [130]. We observe that our approach becomes more potent with larger models as the output diversity increases and the alternatives in the BST become more meaningful. In general, our approach applies to causal language transformers if they (1) provide access to the high-dimensional token-wise embeddings and (2) output the probabilities of the next top-k tokens. While the second requirement is imperative to generate the BST, the first requirement is only needed for the embedding-based widgets.

This means that large parts of our approach are transferable to GPT4 as the current state-of-the-art in causal language modeling. The OpenAI API provides access to the logprobs of the top-k tokens, which can be used to generate the BST. Despite the high-dimensional embeddings not being available for GPT4, the embedding widgets can still be powered from the embeddings produced by other transformers. Sevastjanova et al. [151] and Kehlbeck et al. [8] have studied the embedding spaces of prominent

transformers, suggesting that using the token embeddings of other models might even be beneficial for semantic token analysis.

Transfer of Our Proposed Techniques to Commercial Models — Our approach targets specific user groups. However, we envision some means of explainability embedded into the prominent chat– and completion-based interfaces, like ChatGPT or GitHub Copilot[4]. Currently, ChatGPT only outputs text, and each adaptation has to be triggered by refining the prompt in the hope that the desired output will be generated. This can be frustrating, especially for hallucinated text parts, where no easy solution for editing is available. Here, showing alternative outputs and providing the user with explainability on the likeliness of sequences could bring huge advantages. While GitHub Copilot does show alternatives, those alternatives remain unexplained. Here, showing probabilities or annotating structural elements, c.f., keyword extraction (section 5.3.1) and –coloring W, could further improve the usefulness.

Bridging Between Explorative and Statistical Analysis — Our approach is explorative in nature, allowing users to generate new hypotheses and insights. However, as noted by one of our computer linguist participants, a combination with statistical analysis would be beneficial to validate the generated hypotheses. Therefore, we envision a tighter integration of our approach with statistical analysis tools, e.g., to validate the generated hypotheses with statistical tests. Once this integration is established, annotating the BST branches with statistical metrics could bridge the gap between explorative and statistical analysis. For the current version of the system, we decided against annotating the branches with linguistic metrics to prevent the user from drawing false generalizations from local observations.

Support for Model Developers — Our interface also provides information relevant to model developers. However, for model debugging and refinement, additional tools, e.g., to observe the effects of fine-tuning or investigate common errors in model and data, might be needed.

Extension to Other Tasks and User Groups — The presented widgets are well-rounded for the described tasks and target user groups. However, through an extension with additional widgets, other user groups and tasks and can be addressed, such as machine translation or text summarization. While our approach technically supports these tasks when loading a respective model, additional views and interaction patterns may have to be implemented to optimally support the user and should be part of future research.

Extension to Other Languageges and Prompting Challenges — Due to the prevalence of English training data, most models are known to provide the best performance with English text. We, therefore, focus on English text for the examples and evaluations

[4] `https://github.com/features/copilot`

presented in this work. Since there is no technical limitation and the linguistic phenomena we examine can strongly differ between languages, further languages should be investigated in future work using our proposed approach. We expect that this would reveal interesting insights into the differences between languages and their linguistic phenomena.

Furthermore, the identified and addressed prototypical prompting challenges in section 5.7 represent current areas of active research. Nevertheless, it is likely that there are further interesting linguistic, sociolinguistic, or data- and model-specific prompting challenges that can be investigated using the generAItor workspace.

Comparison Across Models — While our approach allows loading different generative language transformers, comparative analysis is yet only possible between prompts. However, this is not a limitation of our proposed tree-in-the-loop approach and will be implemented in future iterations of the system, enabling additional modes of analysis.

Explainability Instead of Problem Solving — While some of our insights indicate model defects and imply ways to resolve them (e.g., preventing tokenization issues, see 5.7.2.1), this is not the primary focus of our approach. To find tangible ways to resolve such issues, other tools to investigate training data, tokenization, or the deep learning architecture of the model are needed.

5.9 Conclusion

We present the tree-in-the-loop paradigm, putting the beam search tree in the center of the generAItor Visual Analytics technique for language model explainability, comparability, and adaptability. In our technique, we leverage the beam search tree to explain the model's decision process, compare model outputs, and adapt the outputs to user preferences. Enhancing the tree with task-specific widgets creates synergies between the tree and targeted visualizations, interactions, and in-situ explanations. Finally, we provide a three-fold evaluation of our approach. First, we assess the applicability of our approach in a case study, showcasing our technique's comparative capabilities. Particularly, we show how the interplay between the beam search tree and widgets enables new analysis modes, leading to interesting linguistic insights on model biases. Second, we perform two qualitative user studies, the first with six non-experts and the second with four computational linguists, proving the usability of our approach for text generation tasks and linguistic analyses. Also, we quantitatively evaluate the ability to adapt the model to user preferences with relatively few training samples as they arise in our approach. Finally, we tackle five prototypical prompting challenges to highlight how the visual investigation of probabilities and alternative branches aids in verifying and generating hypotheses for LM developers and linguistic researchers alike.

6 Conclusion and Future Work

This chapter concludes the thesis by discussing the presented work, summarizing its key contributions and outlining future research directions. Thereby, it recapitulates how the contributions have addressed the central research question of enabling effective explainability and debugging in deep learning through interactive, visual interfaces. Finally, the chapter discusses the broader implications of this work, reflects on its limitations, and proposes future research directions to address these limitations.

6.1 Summary and Contributions

This thesis has advanced the field of explainable artificial intelligence and debugging for deep learning models by presenting visual and interactive methods grounded in robust theoretical frameworks. The central research question addressed throughout this thesis was how to enable effective explainability and debugging of deep learning models using interactive visual interfaces. This has been explored and answered through conceptual frameworks, system implementations, and extensive evaluations.

The first key contribution, explAIner, presents a novel framework for interactive and explainable machine learning. It introduces an XAI pipeline comprising model understanding, diagnosis, and refinement stages, extended by global monitoring and steering mechanisms. This framework was instantiated in a practical system and validated through a user study, indicating its intuitiveness and potential for integration into daily workflows.

The second major contribution, the iNNspector system, builds upon a comprehensive conceptual framework for systematically debugging deep learning experiments. It categorizes data from machine learning experiments and provides a modular approach to debugging through UI elements and multiple levels of abstraction in model architectures. The implementation of iNNspector demonstrates the practical applicability of this framework, providing a ready-to-use system for systematic debugging of deep learning models. Its evaluation through use cases and a user study with deep learning developers and data scientists underscores its usability, usefulness, and versatility in real-world scenarios.

The third significant contribution is the introduction of the tree-in-the-loop paradigm within the generAItor visual analytics technique. This approach centers on the beam search tree for language model explainability, comparability, and adaptability. The integration of task-specific widgets with the tree enables new analysis modes and provides valuable linguistic insights. The effectiveness of this approach is demonstrated through case studies, qualitative user studies with non-experts and computational linguists, and a quantitative evaluation of model adaptability to user preferences. Furthermore, we show how the generAItor technique can be applied to prototypical prompting challenges, highlighting how the visual investigation of probabilities and alternative branches aids linguistic researchers in probing language models and generating new hypotheses.

The driving research question for the work presented in this thesis was

How to enable effective explainability and debugging of deep learning models using interactive, visual interfaces?

The answers to this question presented in this thesis are multi-faceted. First, the thesis has presented conceptual frameworks for explainable machine learning and systematic debugging of deep learning models, motivated by the need for a structured approach to these processes. They are grounded in state-of-the-art literature and requirements studies with stakeholders, providing a solid theoretical foundation. The frameworks have the potential to guide the development of future systems and provide design guidelines for the integration of explainability and debugging into the machine-learning workflow. Therefore, the frameworks are not only relevant for the systems presented in this thesis but serve as a generalizable foundation for future work in the field. Second, the systems presented in this thesis not only demonstrated the practical applicability of the proposed frameworks but also provided ready-to-use tools for the explainability and debugging of deep learning models. They are designed to be extensible through their modular and open-source architecture, allowing for the integration of new methods and techniques. As our evaluations have shown, the systems are intuitive and easy to use, enabling their integration into daily workflows.

From the presented work, we want to highlight the following key take-home messages.

Structured Approach to Explainability and Debugging — The frameworks presented in this thesis provide structured approaches to explainability and debugging, which is crucial for integrating these processes into the machine-learning workflow. Such structured approaches are needed to guide the development of future systems, tackling the challenge of cluttered interfaces and a variety of systems being scattered across the deep learning ecosystem. explAIner's XAI pipeline (EX1) and iNNspector's debugging framework (IN1, IN2, IN3) provide such structured approaches, which can be used as a foundation for future work in the field.

Data Access in Deep Learning Experiments — A key takeaway from the work presented in this thesis is the importance of making the data arising in deep learning experiments accessible to the user (IN1, GE1). This is not only relevant for deep learning experts, who need to interpret the data to debug their models, but also for non-experts, who want to understand the model's decisions. Notably, "data access" goes beyond showing raw values, as the data needs to be preprocessed, aggregated, and visualized in a way that is useful to the user. iNNspector's way of attaching data to the model's architectural elements and making it visible through tools that can be applied to these elements has shown how this can be achieved even for complex and large-scale data (IN3). As generAItor's tree-in-the-loop approach demonstrated, complex data can be arranged in a way that is intuitive and easy to understand, even for non-experts (GE2). Augmentations, such as generAItor's semantic keyword coloring, can further enhance the readability and interpretability of the data representations (GE4). For this way of accessing data without sacrificing the data's expressiveness, visual interactive interfaces are ideal, as the user can interact with the data to explore it in more detail (GE3).

Modular Design — The systems presented in this thesis are designed to be modular and extensible (IN2, GE5). This applies to both their architecture and their user interfaces. The architectures of the systems allow for the integration of new methods and techniques, which is crucial in a rapidly evolving field such as deep learning and explainable artificial intelligence. The widget-based design of the user interfaces enables the integration of new visualizations and interactions, allowing for the customization of the systems to specific use cases and user groups. As our evaluation of the explAIner system has shown, this flexibility is not only relevant for the visual interactive components but also for the workflow itself. For the subsequent systems iNNspector and generAItor, we have taken this into account, providing a more flexible and iterative workflow, breaking up the phases of understanding, diagnosis, and refinement of the IML loop (EX4).

Diverse Needs for XAI — Our studies with explAIner and generAItor highlight the diverse range of stakeholders in deep learning, each with distinct needs and use cases (EX2, GE6). This spectrum ranges from technical experts like developers who need detailed model insights to interested non-technical users striving to understand and assess the capabilities and limitations of new AI technology. This diversity underscores the need for XAI systems that are not only robust in their technical capabilities but also versatile and accessible. Visualizations and explanations should be comprehensive and informative yet easy to understand and interpret for the respective user group (EX3).

In conclusion, this thesis comprehensively answers the posed research question by developing conceptual frameworks, implementing practical systems, and conducting rigorous evaluations. These contributions collectively enhance the understanding, explainability, and debugging of deep learning models, enabling a more intuitive, systematic, and practical approach to model development, analysis, and refinement. This work not only

pushes the boundaries of practices in the interactive machine-learning workflow but also provides practical tools and methodologies that can be directly applied in various real-world scenarios.

6.2 Broader Impact and Future Research Perspectives

This thesis has made considerable strides in the field of visual interactive techniques for explainable artificial intelligence and deep learning model debugging, and the implications of these advancements have the potential to extend beyond the academic sphere. The frameworks and systems developed herein can guide future developments in the field. Below are key areas where this research could have a broader impact and potential directions for future research.

Making Deep Learning Debugging a Routine Practice — A significant future direction is integrating debugging tools developed in this thesis, such as iNNspector, into popular Integrated Development Environments (IDEs) and deep learning frameworks. With this integration, we aim to push systematic model debugging as a routine part of the deep learning development process, moving away from the typical approach relying on trial and error and best practices. By embedding these tools within the environments where developers already work, the barriers to effective debugging are significantly reduced, thereby streamlining the model development lifecycle.

Integration into Commercial Tools and Interfaces — With the advent of publicly available generative models for image and text generation, such as DALL-E 3 [262] and GPT-3.5, there is a growing need for explainability aimed at interested non-experts. Even a coarse understanding of the underlying principles can significantly enhance the assessability of new technology, thereby simultaneously allaying fears and raising awareness of potential limitations. Therefore, a promising future direction is integrating the tree-in-the-loop approach developed in generAItor into commercial tools and interfaces for non-experts. While the current version of the beam search tree is likely too detailed for casual text generation, an extension of the technique with a simplified tree structure (e.g., to observe uncertainties and alternatives) and additional augmentations (e.g., to detect long-term changes in concepts) could be a valuable addition to such tools.

Comparative Analysis — The comparative analysis features presented in explAIner (see section 3.3.4), iNNspector (see M7 in section 4.3.3 and Multi-Model View [L3] in section 4.4.1), and generAItor (see section 5.4.2) are promising approaches to enable the comparison of models and their predictions. Since comparative analysis is a crucial part of model development, explanation, and debugging, future research should focus on extending these features to enable more detailed and comprehensive comparisons. Regarding explAIner and iNNspector, an extension through juxtaposed views of multiple

model architectures and tools for highlighting differences between them would be a promising direction. For generAItor, this could include functionalities to compare predictions across different models.

Closing the IML Loop — While the systems presented in this thesis prove to be effective in explaining and debugging deep learning models, they are still limited to the passive observation of the model's behavior, leaving the refinement of the model to the user. Therefore, future research should focus on closing the interactive machine learning loop by actively suggesting refinements to the model based on the user's observations. In contrast to automated architecture search, the visual interactive approaches presented in this thesis could enable the user to actively steer the refinement process, thereby ensuring that the model is improved in a way that is meaningful to the user.

Reporting and Collaboration — Another promising direction is the integration of the developed systems with reporting platforms and collaboration tools. This extension would transform individual debugging and explainability tools into collaborative solutions, enabling different stakeholders to develop and reason about deep learning models together. Shared insights and collaborative analysis could enable developers to comprehend the findings of other team members and facilitate the communication of results to non-experts, for example, to build trust in a model's predictions.

Keeping Up with Rapid Advancements — Deep learning and XAI are evolving rapidly, with new methods and techniques emerging constantly. Therefore, a key challenge and area for future research is ensuring that the systems and frameworks developed remain up-to-date and adaptable to incorporate these advancements. Continuous development and updates will be crucial for maintaining the relevance and effectiveness of the proposed approaches and systems.

Adding Support for Other Use Cases — While the system implementations presented in this thesis are built as operational real-world applications, other use cases and application domains are possible. There is considerable scope for adapting and extending the systems to various use cases by adding new explainability techniques to explAIner, debugging tools to iNNspector, and analysis modes to generAItor. By tailoring these tools to specific applications and user needs, their utility and impact can be significantly broadened, facilitating the deployment of AI in diverse domains. In particular, we designed the conceptual frameworks to generalize to other use cases, providing a solid foundation for future work in this direction.

In conclusion, the research presented in this thesis contributes to the academic understanding of explainable AI and deep learning debugging and lays the groundwork for practical, real-world applications. The future research directions outlined above can further advance the field, enabling the development of more effective and usable systems for explainability and debugging.

List of Figures

1.1 Thesis structure overview 5

3.1 Schematic representation of an explainer 28
3.2 Iterative model explanation and refinement workflow 29
3.3 Transitions between model states 30
3.4 Explainer properties 31
3.5 Design overview of the explAIner system 35
3.6 Screenshot of explAIner's diagnosis dashboard 36
3.7 Information cards showing descriptions and results of explainers 38
3.8 Card screenshots of an explAIner use case 40

4.1 Workflow to substantiate, design, implement, and evaluate iNNspector . 52
4.2 Exemplary instantiation of an iNNspector debugging component 66
4.3 iNNspector's global mechanisms to navigate the data space 70
4.4 Screenshot of the iNNspector system frontend 71
4.5 Inspection panel on L3, "Multi-model" 72
4.6 Inspection panel on L2, "Single Model" 73
4.7 Inspection panel on L1, "Neuron-Weight-Network" 74
4.8 Panels, tools, and widgets in the iNNspector UI 75
4.9 iNNspector's widget panel 76
4.10 iNNspector's widget group view 77
4.11 Configuration and navigation panels in iNNspector 78
4.12 Localization and interestingness functionalities in iNNspector 80
4.13 HDF5-structure of iNNspector checkpoint files 83
4.14 UC1: iNNspector widgets to assess model performance 87
4.15 UC2: iNNspector widgets to balance losses in a VAE 88
4.16 UC3: iNNspector widgets to investigate the distribution of activations . 89
4.17 Quantitative study results on the usability of the iNNspector system . . . 92
4.18 Quantitative study results on the usefulness of the iNNspector system . . 93
4.19 Quantitative study results on iNNspector's goal to enable systematic model debugging 95

5.1 generAItor's beam search tree visualization 110
5.2 generAItor widgets for configuration and semantic keyword projection . 111
5.3 The five user tasks supported by generAItor 112

5.4 generAItor's ontology Voronoi treemap and ontological replace menu . . 114
5.5 Exemplary text generation workflow in generAItor 115
5.6 Screenshot of the generAItor workspace in comparative analysis mode . 118
5.7 generAItor's prediction– and keyword embedding pipelines 119
5.8 First example of comparative bias analysis in generAItor 121
5.9 Second example of comparative bias analysis in generAItor 123
5.10 Quantitative evaluation of the generAItor's usability and usefulness . . . 124
5.11 Quantitative evaluation of generAItor's fine-tuning functionality 127
5.12 Ranking the branches of a beam search tree 128
5.13 Scenario for analyzing prompt sensitivity in generAItor 136
5.14 Scenario for analyzing surface form competition in generAItor 136
5.15 Scenario for analyzing negations in generAItor (baseline) 137
5.16 Scenario for analyzing negations in generAItor (negation) 137
5.17 Scenario for analyzing quantifiers in generAItor 137

List of Tables

2.1 Characterization of common XAI methods 10

4.1 Overview of the types of data arising in the DL workflow 61
4.2 iNNspector design dimensions 65

5.1 Example sequences generated in generAItor's comparative mode 122
5.2 Quantitative evaluation of generAItor's fine-tuning functionality 126
5.3 Quantitative evaluation of generAItor's beam search tree 129

List of Listings

4.1 Function to generate the iNNspector header 83

5.1 Algorithm to rank the branches of a beam search tree 128

List of Abbreviations

(V)AE	(Variational) Autoencoder
(X)AI	(Explainable) Artificial Intelligence
(A/D/R)NN	(Artificial/Deep/Recurrent) Neural Network
API	Application Programming Interface
BLOOM	BigScience Large Open-science Open-access Multilingual LM
BST	Beam Search Tree
CAV	Concept Activation Vector
DB	Database
DL	Deep Learning
GAN	Generative Adversarial Network
GPT	Generative Pre-trained Transformer
HDF	Hierarchical Data Format
ID	Identifier
IDE	Integrated Development Environment
IML	Interactive Machine Learning
JSON	JavaScript Object Notation
KL	Kullback-Leibler
LIME	Local Interpretable Model-Agnostic Explanations
(L)LM	(Large) Language Model
LRP	Layer-wise Relevance Propagation
ML	Machine Learning
NLG	Natural Language Generation
NLP	Natural Language Processing
PH	Placeholder
REST	Representational State Transfer
SFC	Surface Form Competition
SHAP	Shapley Additive Explanations
SOTA	State-of-the-art
UI	User Interface
URL	Uniform Resource Locator
VA	Visual Analytics

Bibliography

[1] T. Spinner, U. Schlegel, H. Schafer, and M. El-Assady. “explAIner: A Visual Analytics Framework for Interactive and Explainable Machine Learning”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 1064–1074.

[2] T. Spinner, D. Fürst, and M. El-Assady. “iNNspector: Visual, Interactive Deep Model Debugging”. 2024. arXiv: `2407.17998`.

[3] T. Spinner, R. Kehlbeck, R. Sevastjanova, T. Stähle, D. A. Keim, O. Deussen, and M. El-Assady. “generAItor: Tree-in-the-Loop Text Generation for Language Model Explainability and Adaptation”. In: *ACM Transactions on Interactive Intelligent Systems* 14.2 (2024).

[4] T. Spinner, R. Kehlbeck, R. Sevastjanova, T. Stähle, D. A. Keim, O. Deussen, A. Spitz, and M. El-Assady. “Revealing the Unwritten: Visual Investigation of Beam Search Trees to Address Language Model Prompting Challenges”. 2023. arXiv: `2310.11252`.

[5] T. Spinner, J. Körner, J. Görtler, and O. Deussen. “Towards an Interpretable Latent Space: An Intuitive Comparison of Autoencoders with Variational Autoencoders”. In: *Proceedings of the IEEE VIS Workshop on Visualization for AI Explainability*. 2018.

[6] M. El-Assady, W. Jentner, R. Kehlbeck, U. Schlegel, R. Sevastjanova, F. Sperrle, T. Spinner, and D. Keim. “Towards XAI: Structuring the Processes of Explanations”. In: *ACM CHI Workshop on Human–Centered Machine Learning Perspectives*. 2019.

[7] J. Görtler, T. Spinner, D. Streeb, D. Weiskopf, and O. Deussen. “Uncertainty-Aware Principal Component Analysis”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 822–831.

[8] R. Kehlbeck, R. Sevastjanova, T. Spinner, T. Stähle, and M. El-Assady. “Demystifying the Embedding Space of Language Models”. In: *Proceedings of the IEEE VIS Workshop on Visualization for AI Explainability*. 2021.

[9] T. Spinner, U. Schlegel, M. Schall, F. Sperrle, R. Sevastjanova, B. Gobbo, J. Rauscher, M. El-Assady, and D. A. Keim. “Speculative Execution of Similarity Queries: Real-Time Parameter Optimization through Visual Exploration”. In: *Workshop Proceedings of the EDBT/ICDT Joint Conference*. 2021.

[10] M. El-Assady, R. Kehlbeck, Y. Metz, U. Schlegel, R. Sevastjanova, F. Sperrle, and T. Spinner. “Semantic Color Mapping: A Pipeline for Assigning Meaningful Colors to Text”. In: *IEEE Workshop on Visualization Guidelines in Research, Design, and Education* (2022).

[11] W. Jentner, F. Sperrle, D. Seebacher, M. Kraus, R. Sevastjanova, M. T. Fischer, U. Schlegel, D. Streeb, M. Miller, T. Spinner, E. Cakmak, M. Sharinghousen, P. Meschenmoser, J. Görtler, O. Deussen, F. Stoffel, H. Kabitz, D. A. Keim, M. El-Assady, and J. F. Buchmüller. "Visualisierung der COVID-19-Inzidenzen und Behandlungskapazitäten mit CoronaVis". In: *Resilienz und Pandemie: Handlungsempfehlungen anhand von Erfahrungen mit COVID-19*. Ed. by A. Karsten and S. Voßschmidt. Kohlhammer, 2022. Chap. 2, pp. 176–189.

[12] K. Muhammad, A. Ullah, J. Lloret, J. D. Ser, and V. H. C. de Albuquerque. "Deep Learning for Safe Autonomous Driving: Current Challenges and Future Directions". In: *IEEE Transactions on Intelligent Transportation Systems* 22.7 (2021), pp. 4316–4336.

[13] Y. Hayashi. "Black Box Nature of Deep Learning for Digital Pathology: Beyond Quantitative to Qualitative Algorithmic Performances". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 95–101.

[14] C. Meng, L. Trinh, N. Xu, J. Enouen, and Y. Liu. "Interpretability and fairness evaluation of deep learning models on MIMIC-IV dataset". In: *Scientific Reports* 12.1 (2022).

[15] A. B. Arrieta, N. Díaz-Rodríguez, J. D. Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, and F. Herrera. "Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI". In: *Information Fusion* 58 (2020), pp. 82–115.

[16] JordanTeslaTech. "Incident Number 145". In: *AI Incident Database* (2021). Ed. by S. McGregor.

[17] H. Lakkaraju, S. Tan, J. Adebayo, J. Steinhard, D. Sculley, and R. Caruana. "Debugging Machine Learning Models". Workshop at the International Conference on Learning Representations. 2019.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Curran Associates Inc., 2017, pp. 6000–6010.

[19] "Prepare for truly useful large language models". In: *Nature Biomedical Engineering* 7 (2023), pp. 85–86.

[20] M. T. Ribeiro, S. Singh, and C. Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the ACMSIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1135–1144.

[21] M. T. Ribeiro, S. Singh, and C. Guestrin. "Anchors: High-Precision Model-Agnostic Explanations". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (2018).

[22] B. Kim, M. Wattenberg, J. Gilmer, C. Cai, J. Wexler, F. Viegas, and R. Sayres. "Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)". In: *International Conference on Machine Learning* (2018).

[23] S. Bach, A. Binder, G. Montavon, F. Klauschen, K. Müller, and W. Samek. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PLoS One* 10.7 (2015).

[24] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross. “Towards better understanding of gradient-based attribution methods for Deep Neural Networks”. In: *International Conference on Learning Representations*. OpenReview.net, 2018.

[25] G. Montavon, S. Bach, A. Binder, W. Samek, and Müller, K.R. “Explaining nonlinear classification decisions with deep Taylor decomposition”. In: *Pattern Recognition* 65 (2017), pp. 211–222.

[26] K. Simonyan, A. Vedaldi, and A. Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”. In: *International Conference on Learning Representations*. 2014.

[27] M. Alber, S. Lapuschkin, P. Seegerer, M. Hägele, K. T. Schütt, G. Montavon, W. Samek, K. Müller, S. Dähne, and P. Kindermans. “iNNvestigate neural networks!” 2018. arXiv: `1808.04260`.

[28] A. Shrikumar, P. Greenside, and A. Kundaje. “Learning important features through propagating activation differences”. In: *Proceedings of the International Conference on Machine Learning*. ICML’17. JMLR.org, 2017, pp. 3145–3153.

[29] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization”. In: *IEEE International Conference on Computer Vision*. IEEE, 2017, pp. 618–626.

[30] M. D. Zeiler and R. Fergus. “Visualizing and Understanding Convolutional Networks”. In: *Computer Vision*. Springer International Publishing, 2014, pp. 818–833.

[31] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg. “SmoothGrad: removing noise by adding noise”. 2017. arXiv: `1706.03825`.

[32] M. Sundararajan, A. Taly, and Q. Yan. “Axiomatic Attribution for Deep Networks”. In: *Proceedings of the International Conference on Machine Learning*. JMLR.org, 2017, pp. 3319–3328.

[33] M. Zeiler, D. Krishnan, G. Taylor, and R. Fergus. “Deconvolutional networks”. In: *Conference on Computer Vision and Pattern Recognition*. IEEE, 2010.

[34] A. Harley. “An Interactive Node-Link Visualization of Convolutional Neural Networks”. In: *Advances in Visual Computing*. Springer International Publishing, 2015, pp. 867–877.

[35] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mane, D. Fritz, D. Krishnan, F. B. Viegas, and M. Wattenberg. “Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 1–12.

[36] D. Gunning. *Explainable Artificial Intelligence (XAI) DARPA-BAA-16-53*. Tech. rep. Defense Advanced Research Projects Agency (DARPA), 2016.

[37] A. Adadi and M. Berrada. “Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)”. In: *IEEE Access* 6 (2018), pp. 52138–52160.

[38] C. Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. [Online; accessed 01 Jul 2022]. Lulu.com, 2022.

[39] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. “A Survey of Methods for Explaining Black Box Models”. In: *ACM Computing Surveys* 51.5 (2018).

[40] S. Wachter, B. D. Mittelstadt, and C. Russell. “Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR”. In: *CoRR* abs/1711.00399 (2017). arXiv: 1711.00399.

[41] A. Ghorbani, A. Abid, and J. Zou. “Interpretation of Neural Networks Is Fragile”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (2019), pp. 3681–3688.

[42] A. Dombrowski, M. Alber, C. Anders, M. Ackermann, K. Müller, and P. Kessel. “Explanations can be manipulated and geometry is to blame”. In: *Proceedings of the International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019, pp. 13589–13600.

[43] Z. C. Lipton. “The mythos of model interpretability”. In: *Communications of the ACM* 61.10 (2018), pp. 36–43.

[44] U. Ehsan and M. O. Riedl. “Human-Centered Explainable AI: Towards a Reflective Sociotechnical Approach”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 449–466.

[45] U. Ehsan, P. Wintersberger, Q. V. Liao, E. A. Watkins, C. Manger, H. D. III, A. Riener, and M. O. Riedl. “Human-Centered Explainable AI (HCXAI): Beyond Opening the Black-Box of AI”. In: *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, 2022.

[46] R. Singh, U. Ehsan, M. Cheong, M. O. Riedl, and T. Miller. “LEx: A Framework for Operationalising Layers of Machine Learning Explanations”. In: *CoRR* abs/2104.09612 (2021). arXiv: 2104.09612.

[47] Q. V. Liao, D. Gruen, and S. Miller. “Questioning the AI: Informing Design Practices for Explainable AI User Experiences”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2020.

[48] B. Gobbo, T. Elli, U. Hinrichs, and M. El-Assady. “xai-primer.com -- A Visual Ideation Space of Interactive Explainers”. In: *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, 2022.

[49] A. Smith-Renner, R. Fan, M. Birchfield, T. Wu, J. Boyd-Graber, D. S. Weld, and L. Findlater. “No Explainability without Accountability: An Empirical Study of Explanations and Feedback in Interactive ML”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2020, pp. 1–13.

[50] S. P. Reiss. “The Challenge of Helping the Programmer during Debugging”. In: *Second IEEE Working Conference on Software Visualization*. IEEE, 2014, pp. 112–116.

[51] S. Sacha, M. Kraus, D. Keim, and M. Chen. “VIS4ML: An Ontology for Visual Analytics Assisted Machine Learning”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 385–395.

[52] A. Endert, W. Ribarsky, C. Turkay, B. Wong, I. Nabney, I. Blanco, and F. Rossi. “The State of the Art in Integrating Machine Learning into Visual Analytics”. In: *Computer Graphics Forum* (2017).

[53] J. Yuan, C. Chen, W. Yang, M. Liu, J. Xia, and S. Liu. “A survey of visual analytics techniques for machine learning”. In: *Computational Visual Media* 7.1 (2020), pp. 3–36.

[54] F. Hohman, M. Kahng, R. Pienta, and D. H. Chau. “Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.8 (2018), pp. 2674–2693.

[55] M. Liu, J. Shi, K. Cao, J. Zhu, and S. Liu. “Analyzing the Training Processes of Deep Generative Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 77–87.

[56] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. (Chau. “ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 88–97.

[57] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. “LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 667–676.

[58] S. Liu, X. Wang, M. Liu, and J. Zhu. “Towards Better Analysis of Machine Learning Models: A Visual Analytics Perspective”. In: *Visual Informatics* 1.1 (2017).

[59] D. Smilkov, S. Carter, D. Sculley, and Viégas, F.B. and Wattenberg, M. “Direct-Manipulation Visualization of Deep Networks”. In: *ICML Workshop on Visualization for Deep Learning*. 2016.

[60] M. Kahng, N. Thorat, D. H. P. Chau, F. B. Viegas, and M. Wattenberg. “GAN Lab: Understanding Complex Deep Generative Models using Interactive Visual Experimentation”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 310–320.

[61] P. Rauber, S. Fadel, and Falcão, A.X. and Telea, A.C. “Visualizing the Hidden Activity of Artificial Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 101–110.

[62] A. Bilal, A. Jourabloo, M. Ye, X. Liu, and L. Ren. “Do Convolutional Neural Networks Learn Class Hierarchy?” In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 152–162.

[63] H. Strobelt, S. Gehrmann, M. Behrisch, A. Perer, H. Pfister, and A. M. Rush. “Seq2seq-Vis: A Visual Debugging Tool for Sequence-to-Sequence Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 353–363.

[64] J. Krause, A. Perer, and K. Ng. “Interacting with Predictions: Visual Inspection of Black-box Machine Learning Models”. In: *CHI Conference on Human Factors in Computing Systems*. ACM, 2016.

[65] J. Zhang, Y. Wang, P. Molino, L. Li, and D. Ebert. “Manifold: A Model-Agnostic Framework for Interpretation and Diagnosis of Machine Learning Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 364–373.

[66] K. Cao, M. Liu, H. Su, J. Wu, J. Zhu, and S. Liu. “Analyzing the Noise Robustness of Deep Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 27.7 (2021), pp. 3289–3304.

[67] Y. Ming, S. Cao, R. Zhang, Z. Li, Y. Chen, Y. Song, and H. Qu. “Understanding Hidden Memories of Recurrent Neural Networks”. In: *IEEE Conference on Visual Analytics Science and Technology* (2018).

[68] J. Wang, L. Gou, H. Shen, and H. Yang. “DQNViz: A Visual Analytics Approach to Understand Deep Q-Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 288–298.

[69] X. Zhao, Y. Wu, D. Lee, and W. Cui. “IForest: Interpreting Random Forests via Visual Analytics”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 407–416.

[70] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu. “Towards Better Analysis of Deep Convolutional Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 91–100.

[71] Pezzotti, N. and Hollt, T. and Van Gemert, J. and Lelieveldt, B.P.F. and Eisemann, E. and Vilanova, A. “DeepEyes: Progressive Visual Analytics for Designing Deep Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2017), pp. 98–108.

[72] B. Kwon, M. Choi, J. Kim, E. Choi, Y. Kim, S. Kwon, J. Sun, and J. Choo. “RetainVis: Visual Analytics with Interpretable and Interactive Recurrent Neural Networks on Electronic Medical Records”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2018), pp. 299–309.

[73] S. Murugesan, S. Malik, F. Du, E. Koh, and T. M. Lai. “DeepCompare: Visual and Interactive Comparison of Deep Learning Model Performance”. In: *IEEE Computer Graphics and Applications* 39.5 (2019), pp. 47–59.

[74] M. El-Assady, R. Sevastjanova, F. Sperrle, D. Keim, and C. Collins. “Progressive Learning of Topic Modeling Parameters: A Visual Analytics Framework”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 382–391.

[75] C. Cai, E. Reif, N. Hegde, J. Hipp, B. Kim, D. Smilkov, M. Wattenberg, D. Viegas, G. Corrado, M. Stumpe, and M. Terry. “Human-Centered Tools for Coping with Imperfect Algorithms during Medical Decision-Making”. In: *CHI Conference on Human Factors in Computing Systems* (2019).

[76] M. El-Assady, F. Sperrle, O. Deussen, D. Keim, and C. Collins. “Visual Analytics for Topic Model Optimization based on User-Steerable Speculative Execution”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 374–384.

[77] M. El-Assady, R. Kehlbeck, C. Collins, D. Keim, and O. Deussen. “Semantic Concept Spaces: Guided Topic Model Refinement using Word-Embedding Projections”. In: *IEEE Transactions on Visualization and Computer Graphics* (2019).

[78] P. Y. Simard, S. Amershi, D. M. Chickering, A. E. Pelton, S. Ghorashi, C. Meek, G. Ramos, J. Suh, J. Verwey, M. Wang, and J. Wernsing. “Machine Teaching: A New Paradigm for Building Machine Learning Systems”. 2017. arXiv: `1707.06742`.

[79] J. Krause, A. Dasgupta, J. Swartz, Y. Aphinyanaphongs, and E. Bertini. “A Workflow for Visual Diagnostics of Binary Classifiers using Instance-Level Explanations”. In: *IEEE Conference on Visual Analytics Science and Technology* (2018).

[80] Y. Ming, H. Qu, and E. Bertini. “RuleMatrix: Visualizing and Understanding Classifiers with Rules”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 342–352.

[81] R. Sevastjanova, M. El-Assady, A. Hautli-Janisz, A. Kalouli, R. Kehlbeck, O. Deussen, D. Keim, and M. Butt. “Mixed-Initiative Active Learning for Generating Linguistic Insights in Question Classification”. In: *IEEE VIS Workshop on Data Systems for Interactive Analysis* (2018).

[82] A. Bäuerle, Á. A. Cabrera, F. Hohman, M. Maher, D. Koski, X. Suau, T. Barik, and D. Moritz. “Symphony: Composing Interactive Interfaces for Machine Learning”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2022.

[83] F. Schneider, F. Dangel, and P. Hennig. “Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan. Curran Associates Inc., 2021, pp. 20825–20837.

[84] F. Doshi-Velez and B. Kim. “Towards A Rigorous Science of Interpretable Machine Learning”. 2017. arXiv: 1702.08608.

[85] L. Jiang, S. Liu, and C. Chen. “Recent Research Advances on Interactive Machine Learning”. In: *Journal of Vision* (2018).

[86] G. Hart. “The five w’s of online help systems”. http://www.geoff-hart.com/articles/2002/fivew.htm. [Online; accessed 31 Mar 2019]. 2002.

[87] F. Hutter, L. Kotthoff, and J. Vanschoren, eds. *Automated Machine Learning*. Springer International Publishing, 2019.

[88] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. “Algorithms for Hyper-Parameter Optimization”. In: *Proceedings of the International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2011, pp. 2546–2554.

[89] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *Journal of Machine Learning Research* 18.1 (2017), pp. 6765–6816.

[90] S. Falkner, A. Klein, and F. Hutter. “BOHB: Robust and Efficient Hyperparameter Optimization at Scale”. In: *Proceedings of the International Conference on Machine Learning*. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1437–1446.

[91] S. Thrun and L. Pratt. “Learning to Learn: Introduction and Overview”. In: *Learning to Learn*. Springer US, 1998, pp. 3–17.

[92] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. “Generalizing from a Few Examples”. In: *ACM Computing Surveys* 53.3 (2021), pp. 1–34.

[93] T. Elsken, J. H. Metzen, and F. Hutter. “Neural Architecture Search: A Survey”. In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21.

[94] D. Floreano, P. Dürr, and C. Mattiussi. “Neuroevolution: from architectures to learning”. In: *Evolutionary Intelligence* 1.1 (2008), pp. 47–62.

[95] G. F. Miller, P. M. Todd, and S. U. Hegde. “Designing Neural Networks Using Genetic Algorithms”. In: *Proceedings of the International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1989, pp. 379–384.

[96] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. “Regularized Evolution for Image Classifier Architecture Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (2019), pp. 4780–4789.

[97] B. Zoph and Q. V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *International Conference on Learning Representations*. OpenReview.net, 2017.

[98] J. Bergstra, D. Yamins, and D. Cox. “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures”. In: *Proceedings of the International Conference on Machine Learning*. PMLR, 2013, pp. 115–123.

[99] L. Li and A. Talwalkar. “Random Search and Reproducibility for Neural Architecture Search”. In: *Proceedings of The Uncertainty in Artificial Intelligence Conference*. Proceedings of Machine Learning Research. PMLR, 2020, pp. 367–377.

[100] G. C. Cawley and N. L. Talbot. “On Over-Fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation”. In: *Journal of Machine Learning Research* 11.70 (2010), pp. 2079–2107.

[101] A. Patel. “Tools to Design or Visualize Architecture of Neural Network”. `https://github.com/ashishpatel26/Tools-to-Design-or-Visualize-Architecture-of-Neural-Network`. [Online; accessed 27 Jun 2022]. 2021.

[102] A. LeNail. “NN-SVG: Publication-Ready Neural Network Architecture Schematics”. In: *Journal of Open Source Software* 4.33 (2019), p. 747.

[103] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[104] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (2017). AlexNet, pp. 84–90.

[105] TensorBoard. “TensorBoard: TensorFlow’s visualization toolkit”. `https://www.tensorflow.org/tensorboard/`. [Online; accessed 20 Sep 2021]. 2020.

[106] L. Roeder. “Netron: Visualizer for neural network, deep learning, and machine learning models”. `https://github.com/lutzroeder/netron`. [Online; accessed 30 Apr 2020]. 2022.

[107] Q. Wang, J. Yuan, S. Chen, H. Su, H. Qu, and S. Liu. “Visual Genealogy of Deep Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.11 (2019), pp. 3340–3352.

[108] C. Seifert, A. Aamir, A. Balagopalan, D. Jain, A. Sharma, S. Grottel, and S. Gumhold. “Visualizations of Deep Neural Networks in Computer Vision: A Survey”. In: *Studies in Big Data*. Springer International Publishing, 2017, pp. 123–144.

[109] F. Grün, C. Rupprecht, N. Navab, and F. Tombari. "A Taxonomy and Library for Visualizing Learned Features in Convolutional Neural Networks". In: *CoRR* abs/1606.07757 (2016). arXiv: 1606.07757.

[110] A. Brachmann and C. Redies. "Using Convolutional Neural Network Filters to Measure Left-Right Mirror Symmetry in Images". In: *Symmetry* 8.12 (2016), p. 144.

[111] W. Samek, A. Binder, G. Montavon, S. Lapuschkin, and K. Muller. "Evaluating the Visualization of What a Deep Neural Network Has Learned". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.11 (2017), pp. 2660–2673.

[112] C. Olah, A. Mordvintsev, and L. Schubert. "Feature Visualization". In: *Distill* (2017). [Online; accessed 11 Apr 2022].

[113] F. Hohman, H. Park, C. Robinson, and D. H. (Chau. "Summit: Scaling Deep Learning Interpretability by Visualizing Activation and Attribution Summarizations". In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 1096–1106.

[114] Weights & Biases. "Weights & Biases: Developer-first MLOps Platform". https://wandb.ai. [Online; accessed 21 Nov 2021]. 2021.

[115] J. Wexler, M. Pushkarna, T. Bolukbasi, M. Wattenberg, F. Viegas, and J. Wilson. "The What-If Tool: Interactive Probing of Machine Learning Models". In: *IEEE Transactions on Visualization and Computer Graphics* (2019).

[116] S. Shah, R. Fernandez, and S. Drucker. "A System for Real-Time Interactive Analysis of Deep Learning Training". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. Association for Computing Machinery, 2019.

[117] Microsoft. "Error Analysis: A toolkit to help analyze and improve model accuracy". https://erroranalysis.ai/. [Online; accessed 10 Nov 2021]. 2021.

[118] I. Tenney, J. Wexler, J. Bastings, T. Bolukbasi, A. Coenen, S. Gehrmann, E. Jiang, M. Pushkarna, C. Radebaugh, E. Reif, and A. Yuan. "The Language Interpretability Tool: Extensible, Interactive Visualizations and Analysis for NLP Models". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2020.

[119] Y. Bengio, R. Ducharme, and P. Vincent. "A neural probabilistic language model". In: *Advances in Neural Information Processing Systems* 13 (2000).

[120] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *Cahiers De La Revue De Theologie Et De Philosophie* 323.6088 (1986), pp. 533–536.

[121] D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". 2014. arXiv: 1409.0473.

[122] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. "Transformers: State-of-the-Art Natural Language Processing". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2020, pp. 38–45.

[123] J. Devlin, M. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Ed. by J. Burstein, C. Doran, and T. Solorio. Association for Computational Linguistics, 2019, pp. 4171–4186.

[124] J. Howard and S. Ruder. "Universal Language Model Fine-tuning for Text Classification". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 1 vols. Association for Computational Linguistics, 2018, pp. 328–339.

[125] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. "Language Models are Unsupervised Multitask Learners". OpenAI Blog. 2019.

[126] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Curran Associates Inc., 2020, pp. 1877–1901.

[127] OpenAI et al. "GPT-4 Technical Report". 2023. arXiv: `2303.08774`.

[128] J. Li, T. Tang, W. X. Zhao, and J. Wen. "Pretrained Language Model for Text Generation: A Survey". In: *Proceedings of the International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2021.

[129] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, and M. e. a. Gallé. "BLOOM: A 176B-Parameter Open-Access Multilingual Language Model". 2022. arXiv: `2211.05100`.

[130] Together Computer. "Releasing 3B and 7B RedPajama-INCITE family of models including base, instruction-tuned & chat models". `https://www.together.ai/blog/redpajama-models-v1`. [Online; accessed 2 Sep 2024]. 2023.

[131] M. Danilevsky, K. Qian, R. Aharonov, Y. Katsis, B. Kawas, and P. Sen. "A Survey of the State of Explainable AI for Natural Language Processing". In: *Proceedings of the Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2020, pp. 447–459.

[132] J. Mullenbach, S. Wiegreffe, J. Duke, J. Sun, and J. Eisenstein. "Explainable Prediction of Medical Codes from Clinical Text". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2018.

[133] N. Liu, X. Huang, J. Li, and X. Hu. "On Interpretation of Network Embedding via Taxonomy Induction". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2018.

[134] B. L. Rosa, G. Blasilli, R. Bourqui, D. Auber, G. Santucci, R. Capobianco, E. Bertini, R. Giot, and M. Angelini. "State of the Art of Visual Analytics for eXplainable Deep Learning". In: *Computer Graphics Forum* 42.1 (2023), pp. 319–355.

[135] J. Lee, J. Shin, and J. Kim. "Interactive Visualization and Manipulation of Attention-based Neural Machine Translation". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2017, pp. 121–126.

[136] T. Munz, D. Väth, P. Kuznecov, N. T. Vu, and D. Weiskopf. "Visualization-based improvement of neural machine translation". In: *Computers & Graphics* 103 (2022), pp. 45–60.

[137] H. Strobelt, J. Kinley, R. Krueger, J. Beyer, H. Pfister, and A. M. Rush. "GenNI: Human-AI Collaboration for Data-Backed Text Generation". In: *IEEE Transactions on Visualization and Computer Graphics* 28.1 (2022), pp. 1106–1116.

[138] C. Chen, K. Lin, and D. Klein. "Constructing Taxonomies from Pretrained Language Models". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2021.

[139] J. Huang, Y. Xie, Y. Meng, Y. Zhang, and J. Han. "CoRel: Seed-Guided Topical Taxonomy Construction by Concept Learning and Relation Transferring". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2020.

[140] C. Zhang, F. Tao, X. Chen, J. Shen, M. Jiang, B. Sadler, M. Vanni, and J. Han. "TaxoGen". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2018.

[141] M. Jiang, X. Song, J. Zhang, and J. Han. "TaxoEnrich: Self-Supervised Taxonomy Completion via Structure-Semantic Representations". In: *Proceedings of the ACM Web Conference*. ACM, 2022.

[142] H. Xu, Y. Chen, Z. Liu, Y. Wen, and X. Yuan. "TaxoPrompt: A Prompt-based Generation Method with Taxonomic Context for Self-Supervised Taxonomy Expansion". In: *Proceedings of the International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2022.

[143] Y. Xiang, Z. Zhang, J. Chen, X. Chen, Z. Lin, and Y. Zheng. "OntoEA: Ontology-guided Entity Alignment via Joint Knowledge Graph Embedding". In: *Findings of the Association for Computational Linguistics: ACL-IJCNLP*. Association for Computational Linguistics, 2021.

[144] Y. Tan, C. Yang, X. Wei, C. Chen, L. Li, and X. Zheng. "Enhancing Recommendation with Automated Tag Taxonomy Construction in Hyperbolic Space". In: *IEEE International Conference on Data Engineering*. IEEE, 2022.

[145] Z. Li, Y. Wang, X. Yan, W. Meng, Y. Li, and J. Yang. "TaxoTrans". In: *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2022.

[146] B. Scarlini, T. Pasini, and R. Navigli. "With More Contexts Comes Better Performance: Contextualized Sense Embeddings for All-Round Word Sense Disambiguation". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2020.

[147] S. Conia and R. Navigli. “Conception: Multilingually-Enhanced, Human-Readable Concept Vector Representations”. In: *Proceedings of the International Conference on Computational Linguistics*. International Committee on Computational Linguistics, 2020.

[148] R. Navigli and S. P. Ponzetto. “BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network”. In: *Artificial Intelligence* 193 (2012), pp. 217–250.

[149] A. Rogers, O. Kovaleva, and A. Rumshisky. “A Primer in BERTology: What We Know About How BERT Works”. In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 842–866.

[150] K. Ethayarajh. “How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Proceedings and the International Joint Conference on Natural Language Processing*. ACL, 2019, pp. 55–65.

[151] R. Sevastjanova, A. Kalouli, C. Beck, H. Hauptmann, and M. El-Assady. “LMFingerprints: Visual Explanations of Language Model Embedding Spaces through Layerwise Contextualization Scores”. In: *Computer Graphics Forum* 41.3 (2022), pp. 295–307.

[152] E. Reif, A. Yuan, M. Wattenberg, F. B. Viegas, A. Coenen, A. Pearce, and B. Kim. “Visualizing and Measuring the Geometry of BERT”. In: *Advances in Neural Information Processing Systems*. Curran Associates Inc., 2019, pp. 8594–8603.

[153] G. Wiedemann, S. Remus, A. Chawla, and C. Biemann. “Does BERT Make Any Sense? Interpretable Word Sense Disambiguation with Contextualized Embeddings”. In: *Proceedings of the Conference on Natural Language Processing*. German Society for Computational Linguistics & Language Technology, 2019, pp. 161–170.

[154] A. Gatt and E. Krahmer. “Survey of the State of the Art in Natural Language Generation: Core tasks, applications and evaluation”. In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 65–170.

[155] L. Qin, V. Shwartz, P. West, C. Bhagavatula, J. D. Hwang, R. L. Bras, A. Bosselut, and Y. Choi. “Back to the Future: Unsupervised Backprop-based Decoding for Counterfactual and Abductive Commonsense Reasoning”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2020.

[156] S. Dathathri, A. Madotto, J. Lan, J. Hung, E. Frank, P. Molino, J. Yosinski, and R. Liu. “Plug and Play Language Models: A Simple Approach to Controlled Text Generation”. 2019. arXiv: 1912.02164.

[157] Z. Hu, Z. Yang, X. Liang, R. Salakhutdinov, and E. P. Xing. “Toward Controlled Generation of Text”. In: *Proceedings of the International Conference on Machine Learning*. Ed. by D. Precup and Y. W. Teh. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1587–1596.

[158] X. He. “Parallel Refinements for Lexically Constrained Text Generation with BART”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021.

[159] X. Hua and L. Wang. "PAIR: Planning and Iterative Refinement in Pre-trained Transformers for Long Text Generation". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2020.

[160] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020, pp. 7871–7880.

[161] H. Zhang, H. Song, S. Li, M. Zhou, and D. Song. "A Survey of Controllable Text Generation Using Transformer-based Pre-trained Language Models". In: *ACM Computing Surveys* 56.3 (2023), pp. 1–37.

[162] W. Yu, C. Zhu, Z. Li, Z. Hu, Q. Wang, H. Ji, and M. Jiang. "A Survey of Knowledge-enhanced Text Generation". In: *ACM Computing Surveys* 54.11s (2022), pp. 1–38.

[163] W. Du, Z. M. Kim, V. Raheja, D. Kumar, and D. Kang. "Read, Revise, Repeat: A System Demonstration for Human-in-the-loop Iterative Text Revision". In: *Proceedings of the First Workshop on Intelligent and Interactive Writing Assistants*. Association for Computational Linguistics, 2022.

[164] V. Padmakumar and H. He. "Machine-in-the-Loop Rewriting for Creative Image Captioning". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2022.

[165] S. Gehrmann, H. Strobelt, R. Kruger, H. Pfister, and A. M. Rush. "Visual Interaction with Deep Learning Models through Collaborative Semantic Inference". In: *IEEE Transactions on Visualization and Computer Graphics* (2019).

[166] A. Yuan, A. Coenen, E. Reif, and D. Ippolito. "Wordcraft: Story Writing With Large Language Models". In: *International Conference on Intelligent User Interfaces*. ACM, 2022.

[167] I. Garrido-Muñoz, A. Montejo-Ráez, F. Martínez-Santiago, and L. A. Ureña-López. "A survey on bias in deep NLP". In: *Applied Sciences* 11.7 (2021), p. 3184.

[168] S. L. Blodgett, S. Barocas, H. Daumé III, and H. Wallach. "Language (Technology) is Power: A Critical Survey of "Bias" in NLP". In: *Proceedings of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020, pp. 5454–5476.

[169] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan. "A survey on bias and fairness in machine learning". In: *ACM Computing Surveys* 54.6 (2021), pp. 1–35.

[170] A. Caliskan, J. J. Bryson, and A. Narayanan. "Semantics derived automatically from language corpora contain human-like biases". In: *Science* 356.6334 (2017), pp. 183–186.

[171] P. P. Liang, C. Wu, L. Morency, and R. Salakhutdinov. "Towards Understanding and Mitigating Social Biases in Language Models". In: *Proceedings of the International Conference on Machine Learning*. PMLR, 2021, pp. 6565–6576.

[172] M. Nadeem, A. Bethke, and S. Reddy. “StereoSet: Measuring stereotypical bias in pretrained language models”. In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2021, pp. 5356–5371.

[173] S. Alnegheimish, A. Guo, and Y. Sun. “Using Natural Sentence Prompts for Understanding Biases in Language Models”. In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2022, pp. 2824–2830.

[174] W. McCulloch and W. Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* (1943).

[175] W. Rawat and Z. Wang. “Deep convolutional neural networks for image classification: A comprehensive review”. In: *Neural computation* (2017).

[176] T. Young, D. Hazarika, S. Poria, and E. Cambria. “Recent Trends in Deep Learning Based Natural Language Processing”. In: *IEEE Computational Intelligence Magazine* 13.3 (2018), pp. 55–75.

[177] F. Chollet et al. “Keras”. `https://keras.io`. [Online; accessed 2 Sep 2024]. 2015.

[178] P. Voosen. “How AI detectives are cracking open the black box of deep learning”. In: *Science* (2017).

[179] S. Young, D. Rose, T. Karnowski, S. Lim, and R. Patton. “Optimizing Deep Learning Hyper-parameters Through an Evolutionary Algorithm”. In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2015.

[180] K. Siau and W. Wang. “Building Trust in Artificial Intelligence, Machine Learning, and Robotics”. In: *Cutter Business Technology Journal* (2018).

[181] European Union. “European General Data Protection Regulation”. `https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en`. [Online; accessed 28 Mar 2019]. 2018.

[182] L. Kaastra and B. Fisher. “Field experiment methodology for pair analytics”. In: *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*. ACM, 2014.

[183] D. M. W. Powers. “Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation”. In: *Journal of Machine Learning Technologies* (2008).

[184] T. G. Dietterich and E. B. Kong. *Machine Learning Bias, Statistical Bias, and Statistical Variance of Decision Tree Algorithms*. Tech. rep. Technical report, Department of Computer Science, Oregon State University, 1995.

[185] D. P. Solomatine and D. L. Shrestha. “A novel method to estimate model uncertainty using machine learning techniques”. In: *Water Resources Research* (2009).

[186] J. Quionero-Candela, M. Sugiyama, A. Schwaighofer, and N. Lawrence. *Dataset Shift in Machine Learning*. The MIT Press, 2009.

[187] S. Geman, E. Bienenstock, and R. Doursat. “Neural Networks and the Bias/Variance Dilemma”. In: *Neural Computation* 4.1 (1992), pp. 1–58.

[188] F. Sperrle, J. Bernard, M. Sedlmair, D. Keim, and M. El-Assady. “Speculative Execution for Guided Visual Analytics”. In: *IEEE VIS Workshop for Machine Learning from User Interaction for Visualization and Analytics*. IEEE, 2018.

[189] J. Fails and D. Olsen. “Interactive machine learning”. In: *ACM Conference on Intelligent User Interfaces*. ACM, 2003.

[190] G. Malkomes, C. Schaff, and R. Garnett. “Bayesian optimization for automated model selection”. In: Curran Associates Inc., 2016, pp. 2900–2908.

[191] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, E. Horvitz, P. Allen, and A. Four-ney. “Guidelines for Human-AI Interaction”. In: *CHI Conference on Human Factors in Computing Systems* (2019).

[192] R. Sevastjanova, F. Beck, B. Ell, C. Turkay, R. Henkin, M. Butt, D. Keim, and M. El-Assady. “Going beyond Visualization: Verbalization as Complementary Medium to Explain Machine Learning Models”. In: *Proceedings of the IEEE VIS Workshop on Visualization for AI Explainability*. 2018.

[193] A. Gelman and E. Loken. “The garden of forking paths: Why multiple comparisons can be a problem, even when there is no “fishing expedition” or “p-hacking” and the research hypothesis was posited ahead of time”. In: *Department of Statistics, Columbia University* (2013).

[194] H. Stitz, S. Luger, M. Streit, and N. Gehlenborg. “AVOCADO: Visualization of Workflow-Derived Data Provenance for Reproducible Biomedical Research”. In: *Computer Graphics Forum* (2016).

[195] S. Chen, J. Li, G. Andrienko, N. Andrienko, Y. Wang, P. Nguyen, and C. Turkay. “Supporting Story Synthesis: Bridging the Gap between Visual Analytics and Storytelling”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.7 (2018), pp. 2499–2516.

[196] W. Jentner, R. Sevastjanova, F. Stoffel, D. A. Keim, J. Bernard, and M. El-Assady. “Minions, Sheep, and Fruits: Metaphorical Narratives to Explain Artificial Intelligence and Build Trust”. In: *Proceedings of the IEEE VIS Workshop on Visualization for AI Explainability*. 2018.

[197] F. Sperrle, A. Jeitler, J. Bernard, D. A. Keim, and M. El-Assady. “Learning and Teaching in Co-Adaptive Guidance for Mixed-Initiative Visual Analytics”. In: *EuroVis Workshop on Visual Analytics* (2020).

[198] E. W. Dijkstra. “On the Role of Scientific Thought”. In: *Selected Writings on Computing: A personal Perspective*. Springer New York, 1982, pp. 60–66.

[199] TensorFlow. “Developing a TensorBoard plugin”. `https://github.com/tensorflow/tensorboard-plugin-example`. [Online; accessed 18 Jun 2019]. 2019.

[200] H. Bischof, A. Pinz, and W. G. Kropatsch. “Visualization methods for neural networks”. In: *Proceedings of the IAPR International Conference on Pattern Recognition*. IEEE Computer Society Press, 1992, pp. 581–585.

[201] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. "Deep Reinforcement Learning that Matters". In: *AAAI Conference on Artificial Intelligence* (2018).

[202] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia. "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers". In: *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013.

[203] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. "Understanding Neural Networks Through Deep Visualization". In: *ICML Deep Learning Workshop*. 2015.

[204] A. A. Cabrera, W. Epperson, F. Hohman, M. Kahng, J. Morgenstern, and D. H. Chau. "FAIRVIS: Visual Analytics for Discovering Intersectional Bias in Machine Learning". In: *IEEE Conference on Visual Analytics Science and Technology*. IEEE, 2019.

[205] R. Kuprieiev, skshetry, D. Petrov, P. Redzyński, C. da Costa-Luis, P. Rowlands, A. Schepanovski, I. Shcheklein, B. Taskaya, J. Orpinel, D. de la Iglesia Castro, Gao, F. Santos, A. Sharma, Zhanibek, D. Hodovic, N. Kodenko, A. Grigorev, Earl, D. Berenbaum, N. Dash, G. Vyshnya, daniele, maykulkarni, M. Hora, Vera, S. Mangal, and W. Baranowski. "DVC: Data Version Control - Git for Data & Models". In: *Zenodo* (2022).

[206] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. "Going deeper with convolutions". In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2015.

[207] J. Frankle and M. Carbin. "The Lottery Ticket Hypothesis: Training Pruned Neural Networks". In: *CoRR* abs/1803.03635 (2018). arXiv: 1803.03635.

[208] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989). LeNet, pp. 541–551.

[209] L. Geng and H. J. Hamilton. "Interestingness Measures for Data Mining: A Survey". In: *ACM Computing Surveys* 38.3 (2006), p. 9.

[210] R. Sharma, M. Kaushik, S. A. Peious, S. B. Yahia, and D. Draheim. "Expected vs. Unexpected: Selecting Right Measures of Interestingness". In: *Big Data Analytics and Knowledge Discovery*. Springer International Publishing, 2020, pp. 38–47.

[211] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI'17. AAAI Press, 2017, pp. 4278–4284.

[212] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2016.

[213] D. P. Kingma and M. Welling. "Auto-Encoding Variational Bayes". In: *International Conference on Learning Representations*. 2014.

[214] F. Sperrle, M. El-Assady, G. Guo, R. Borgo, D. H. Chau, A. Endert, and D. Keim. "A Survey of Human-Centered Evaluations in Human-Centered Machine Learning". In: *Computer Graphics Forum* 40.3 (2021), pp. 543–568.

[215] R. Arias-Hernandez, L. T. Kaastra, T. M. Green, and B. Fisher. "Pair Analytics: Capturing Reasoning Processes in Collaborative Visual Analytics". In: *Hawaii International Conference on System Sciences*. IEEE, 2011.

[216] S. G. Hart and L. E. Staveland. "Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research". In: *Advances in Psychology*. Elsevier, 1988, pp. 139–183.

[217] J. Brooke. "SUS – a quick and dirty usability scale". In: Taylor & Francis, 1996, pp. 189–194.

[218] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. "Vega-Lite: A Grammar of Interactive Graphics". In: *IEEE Transactions on Visualization and Computer Graphics* (2017).

[219] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Computer Aided Verification*. Springer International Publishing, 2017, pp. 97–117.

[220] A. V. Looveren and J. Klaise. "Interpretable Counterfactual Explanations Guided by Prototypes". In: *Machine Learning and Knowledge Discovery in Databases*. Springer International Publishing, 2021, pp. 650–665.

[221] I. Goodfellow, J. Shlens, and C. Szegedy. "Explaining and Harnessing Adversarial Examples". In: *International Conference on Learning Representations*. 2015.

[222] N. Carlini and D. Wagner. "Towards Evaluating the Robustness of Neural Networks". In: *IEEE Symposium on Security and Privacy*. IEEE, 2017.

[223] E. Saldanha, B. Praggastis, T. Billow, and D. L. Arendt. "ReLVis: Visual Analytics for Situational Awareness During Reinforcement Learning Experimentation". In: *EuroVis Short Papers* (2019).

[224] Y. Metz, U. Schlegel, D. Seebacher, M. El-Assady, and D. Keim. "A Comprehensive Workflow for Effective Imitation and Reinforcement Learning with Visual Analytics". In: *EuroVis Workshop on Visual Analytics*. The Eurographics Association, 2022.

[225] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung. "Survey of Hallucination in Natural Language Generation". In: *ACM Computing Surveys* 55.12 (2023), pp. 1–38.

[226] D. Alba. "OpenAI Chatbot Spits Out Biased Musings, Despite Guardrails". In: *Bloomberg* (2022).

[227] M. El-Assady, R. Sevastjanova, D. Keim, and C. Collins. "ThreadReconstructor: Modeling Reply-Chains to Untangle Conversational Text through Visual Analytics". In: *Computer Graphics Forum* 37.3 (2018), pp. 351–365.

[228] C. Metz. "The New Chatbots Could Change the World. Can You Trust Them?" In: *New York Times* (2022).

[229] K. Roose. "How Chatbots and Large Language Models, or LLMs, Actually Work". In: *New York Times* (2023).

[230] R. J. Sternberg and K. Sternberg. *Cognitive Psychology*. Nelson Education, 2016.

[231] Y. LeCun. “Do Language Models Need Sensory Grounding for Meaning and Understanding?” The Philosophy of Deep Learning. 2023.

[232] A. Lauscher, T. Lueken, and G. Glavaš. “Sustainable Modular Debiasing of Language Models”. In: *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2021, pp. 4782–4797.

[233] S. Mishra, D. Khashabi, C. Baral, and H. Hajishirzi. “Cross-Task Generalization via Natural Language Crowdsourcing Instructions”. In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 1 vols. Association for Computational Linguistics, 2022.

[234] P. von Platen. “How to generate text: using different decoding methods for language generation with Transformers”. `https://huggingface.co/blog/how-to-generate`. [Online; accessed 29 Mar 2023]. 2020.

[235] J. Hartmann, M. Heitmann, C. Schamp, and O. Netzer. “The Power of Brand Selfies”. In: *Journal of Marketing Research* (2021).

[236] A. Radford, J. Wu, D. Amodei, D. Amodei, J. Clark, M. Brundage, and I. Sutskever. “Better Language Models and Their Implications”. `https://openai.com/blog/better-language-models/`. [Online; accessed 18 Mar 2021]. 2019.

[237] L. McInnes, J. Healy, N. Saul, and L. Grossberger. “UMAP: Uniform Manifold Approximation and Projection”. In: *The Journal of Open Source Software* 3.29 (2018), p. 861.

[238] A. Williams, N. Nangia, and S. Bowman. “A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference”. In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2018, pp. 1112–1122.

[239] A. J. Teuling, R. Stöckli, and S. I. Seneviratne. “Bivariate colour maps for visualizing climate data”. In: *International Journal of Climatology* 31.9 (2010), pp. 1408–1412.

[240] Deep NLP. “Bias in NLP”. `https://github.com/cisnlp/bias-in-nlp`. [Online; accessed 15 Nov 2023]. 2023.

[241] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister. “UpSet: Visualization of Intersecting Sets”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 1983–1992.

[242] K. Lu, P. Mardziel, F. Wu, P. Amancharla, and A. Datta. “Gender bias in neural natural language processing”. In: *Logic, Language, and Security: Essays Dedicated to Andre Scedrov on the Occasion of His 65th Birthday* (2020), pp. 189–202.

[243] R. Campos, V. Mangaravite, A. Pasquali, A. Jorge, C. Nunes, and A. Jatowt. “YAKE! Keyword extraction from single documents using multiple local features”. In: *Information Sciences* 509 (2020), pp. 257–289.

[244] M. Steiger, J. Bernard, S. Thum, S. Mittelstädt, M. Hutter, D. A. Keim, and J. Kohlhammer. “Explorative analysis of 2D color maps”. In: *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. 2015, pp. 151–160.

[245] N. Corver and H. van Riemsdijk. *Semi-lexical Categories: The Function of Content Words and the Content of Function Words*. De Gruyter Mouton, 2001.

[246] A. Kalouli, R. Sevastjanova, C. Beck, and M. Romero. "Negation, Coordination, and Quantifiers in Contextualized Language Models". In: *Proceedings of the International Conference on Computational Linguistics*. International Committee on Computational Linguistics, 2022, pp. 3074–3085.

[247] R. Paulus, C. Xiong, and R. Socher. "A Deep Reinforced Model for Abstractive Summarization". In: *CoRR* abs/1705.04304 (2017). arXiv: 1705.04304.

[248] I. Loshchilov and F. Hutter. "Fixing Weight Decay Regularization in Adam". In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101.

[249] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. "Learning Word Vectors for Sentiment Analysis". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2011, pp. 142–150.

[250] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. "Perplexity—a measure of the difficulty of speech recognition tasks". In: *The Journal of the Acoustical Society of America* 62.S1 (1977), pp. 63–63.

[251] A. Webson and E. Pavlick. "Do Prompt-Based Models Really Understand the Meaning of Their Prompts?" In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2022, pp. 2300–2344.

[252] S. Gehman, S. Gururangan, M. Sap, Y. Choi, and N. A. Smith. "RealToxicityPrompts: Evaluating Neural Toxic Degeneration in Language Models". In: *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2020.

[253] M. Domingo, M. García-Martínez, A. Helle, F. Casacuberta, and M. Herranz. "How Much Does Tokenization Affect Neural Machine Translation?" In: *Lecture Notes in Computer Science*. Springer Nature Switzerland, 2023, pp. 545–554.

[254] A. Holtzman, P. West, V. Shwartz, Y. Choi, and L. Zettlemoyer. "Surface Form Competition: Why the Highest Probability Answer Isn't Always Right". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021.

[255] N. Kassner and H. Schütze. "Negated and Misprimed Probes for Pretrained Language Models: Birds Can Talk, But Cannot Fly". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020, pp. 7811–7818.

[256] D. Summers-Stay, C. Bonial, and C. Voss. "What Can a Generative Language Model Answer About a Passage?" In: *Proceedings of the Workshop on Machine Reading for Question Answering*. Association for Computational Linguistics, 2021, pp. 73–81.

[257] T. H. Truong, T. Baldwin, K. Verspoor, and T. Cohn. "Language models are not naysayers: An analysis of language models on negation benchmarks". 2023. arXiv: 2306.08189.

[258] A. Warstadt, Y. Cao, I. Grosu, W. Peng, H. Blix, Y. Nie, A. Alsop, S. Bordia, H. Liu, A. Parrish, S. Wang, J. Phang, A. Mohananey, P. M. Htut, P. Jeretic, and S. R. Bowman. "Investigating BERT's Knowledge of Language: Five Analysis Methods with NPIs". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2019, pp. 2877–2887.

[259] A. Gupta. "Probing Quantifier Comprehension in Large Language Models". 2023. arXiv: 2306.07384.

[260] S. Husse and A. Spitz. "Mind Your Bias: A Critical Review of Bias Detection Methods for Contextual Language Models". In: *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2022.

[261] R. Sevastjanova and M. El-Assady. "Beware the Rationalization Trap! When Language Model Explainability Diverges from our Mental Models of Language". In: *Communication in Human-AI Interaction Workshop at IJCAI-ECAI* (2022).

[262] J. Betker, G. Goh, L. Jing, T. Brooks, J. Wang, L. Li, L. Ouyang, J. Zhuang, J. Lee, Y. Guo, et al. "Improving image generation with better captions". In: *Computer Science* (2023).